# [Analysez vos données textuelles](https://openclassrooms.com/fr/courses/4470541-analysez-vos-donnees-textuelles)

Table des matières

[Analysez vos données textuelles 1](#_Toc57895088)

[**Récupérez et explorez le corpus de textes** 3](#_Toc57895089)

[**Récupération du corpus de texte** 3](#_Toc57895090)

[**Exploration du texte : tokenisation et analyse des fréquences** 5](#_Toc57895091)

[**Conclusion** 9](#_Toc57895092)

[Nettoyez et normalisez les données 10](#_Toc57895093)

[Première passe de nettoyage : supprimer les stopwords 10](#_Toc57895094)

[Deuxième passe : lemmatisation ou racinisation (stemming) 12](#_Toc57895095)

[Conclusion 13](#_Toc57895096)

[**Entraînez-vous à prétraiter un corpus en vue de créer un moteur de résumés** 14](#_Toc57895097)

[**À vous de jouer** 14](#_Toc57895098)

[**Représentez votre corpus en "bag of words"** 16](#_Toc57895099)

[**Qu'est-ce qu'un « bag of words »** 16](#_Toc57895100)

[**Prendre en compte les co-occurences** 16](#_Toc57895101)

[**Une autre manière de pondérer : le tf-idf** 17](#_Toc57895102)

[**Extraire les informations** 19](#_Toc57895103)

[**Attention aux matrices creuses** 20](#_Toc57895104)

[**Conclusion** 21](#_Toc57895105)

[**Effectuez des plongements de mots (word embeddings)** 22](#_Toc57895106)

[**Utiliser un modèle de langage** 25](#_Toc57895107)

[**Alternatives à word2vec** 25](#_Toc57895108)

[**Entraîner son propre embedding** 25](#_Toc57895109)

[**Conclusion** 26](#_Toc57895110)

[Modélisez des sujets avec des méthodes non supervisées 27](#_Toc57895111)

[Première intuition : Latent Dirichlet Allocation (LDA) 28](#_Toc57895112)

[Laissez la partie inférence aux librairies ! 30](#_Toc57895113)

[Utiliser LDA sur un cas pratique : Le newsgroup dataset 30](#_Toc57895114)

[Conclusion 33](#_Toc57895115)

[Partie 2 34](#_Toc57895116)

[Compétences évaluées 34](#_Toc57895117)

[ Classifier des textes avec une LD 34](#_Toc57895118)

[ Question 1 34](#_Toc57895119)

[ Question 2 34](#_Toc57895120)

[ Question 3 35](#_Toc57895121)

[ Question 4 36](#_Toc57895122)

[ Question 5 36](#_Toc57895123)

[ Question 6 37](#_Toc57895124)

[ Question 7 37](#_Toc57895125)

[ Question 8 38](#_Toc57895126)

[ Question 9 38](#_Toc57895127)

[Opérez une première classification naïve de sentiments 39](#_Toc57895128)

[Petit rappel de la méthodologie 39](#_Toc57895129)

[Rappel sur les algorithmes de classification 40](#_Toc57895130)

[Evaluation du sentiment d’un corpus de commentaires Amazon 40](#_Toc57895131)

[Conclusion 44](#_Toc57895132)

[Allez plus loin dans la classification de mots 45](#_Toc57895133)

[Particularités de la classification de sentiments 45](#_Toc57895134)

[Utiliser d’autres types d'algorithmes de classification supervisée 45](#_Toc57895135)

[Conclusion 46](#_Toc57895136)

[Traitez le corpus de textes à l'aide de réseaux de neurones 47](#_Toc57895137)

[Représentation des features 47](#_Toc57895138)

[Différentes architectures possibles 47](#_Toc57895139)

[Etat de l’art des réseaux de neurones en traitement du langage 49](#_Toc57895140)

[Applications 50](#_Toc57895141)

[Une petite application illustrative : retour aux chanteurs de rap 51](#_Toc57895142)

[Conclusion 52](#_Toc57895143)

[**Entraînez-vous à classifier du texte** 53](#_Toc57895144)

[**À vous de jouer** 53](#_Toc57895145)

Bienvenue dans ce cours de traitement du langage naturel. L’objectif de ce cours est de comprendre les méthodes qui permettent de **transformer le texte** en features exploitables par des algorithmes de machine learning, et les architectures et modèles qui correspondent le mieux à ce type de données. En l’occurence un ensemble de documents texte **non-structurés.**

Ce cours est divisé en 3 parties : une première qui traite de l'exploration, du **nettoyage et de la normalisation du texte**. Une seconde partie dédiée **aux différents types de transformations** qui vont vous permettre de mieux comprendre vos données textuelles et de **créer des features** que vous pourrez utiliser dans vos algorithmes de machine learning. La dernière partie sera consacrée à la **classification du texte** à l'aide de l'apprentissage automatique sous forme de **réseau de neurones**.

**Objectifs pédagogiques :**

* Effectuez un pré-traitement de corpus de texte
* Maîtrisez les techniques de bag-of-words et de plongements de mots (*word embeddings)*
* Modélisez des sujets de manière non-supervisée (LDA, etc.)
* Classer des corpus de texte avec des méthodes supervisées (réseaux de neurones)

**Prérequis :**

Ce cours se situe au croisement des mathématiques et de l'informatique. Pour en profiter pleinement, n'hésitez pas à vous rafraîchir la mémoire, avant ou pendant le cours, sur :

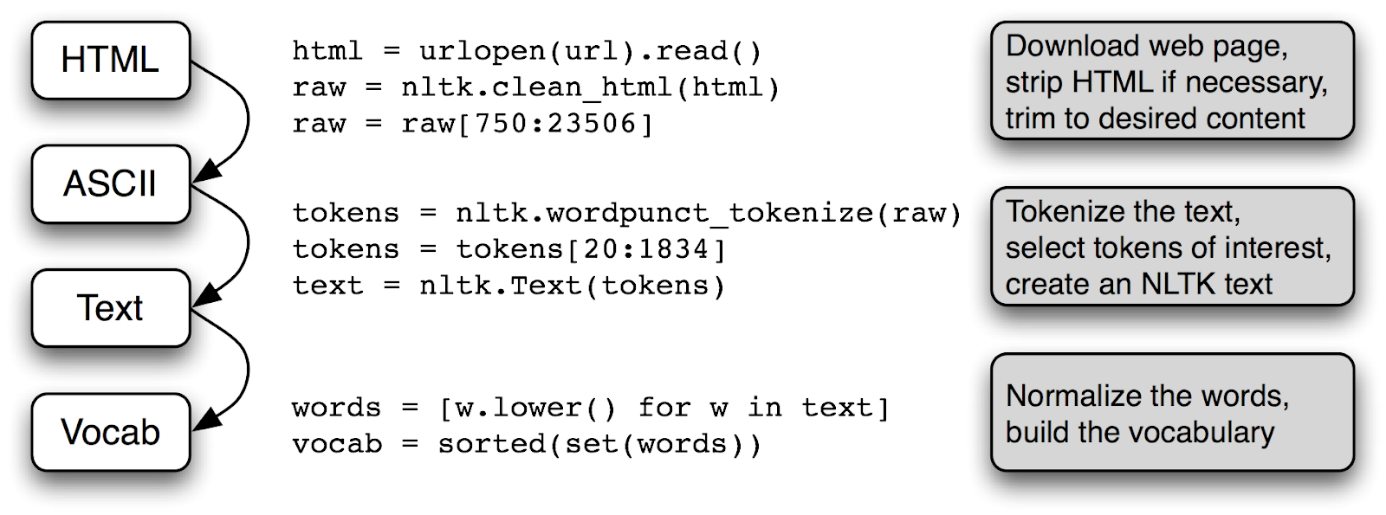
* [**Python**](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science) pour le calcul numérique (numpy) et la création de graphiques (pyplot), que nous utiliserons dans les parties TP du cours,
* Quelques notions d'**algèbre linéaire** : manipulation de vecteurs, multiplications de matrices, normes, et valeurs/vecteurs propres,
* Quelques notions de **probabilités** et **statistiques**, telles que distribution de loi de probabilité et variance,
* [**Les modèles non-supervisées**](https://openclassrooms.com/courses/explorez-vos-donnees-avec-des-algorithmes-non-supervises) permettront de modéliser des features automatiquement à partir du texte
* [**Les modèles supervisées non-linéaires**](https://openclassrooms.com/courses/utilisez-des-modeles-supervises-non-lineaires)sont indispensables au traitement du texte, notamment les réseaux de neurones séquentiels

**Récupérez et explorez le corpus de textes**

La première étape du traitement des données texte est de récupérer le texte et le nettoyer afin de pouvoir l'utiliser ultérieurement dans vos algorithmes. C'est aussi bien de passer par une petite exploration afin de mieux le comprendre. 🔬

Le pré-traitement du texte comprend les étapes suivantes :

1. Récupération du **corpus**par scraping ou en téléchargeant des fichiers textes, par exemple. Cela peut demander l'utilisation de regex afin de récupérer uniquement les parties qui vous intéressent.
2. La **tokenization,** qui désigne le découpage en mots des différents documents qui constituent votre corpus
3. La **normalisation** et la construction du dictionnaire qui permet de ne pas prendre en compte des détails importants au niveau local (ponctuation, majuscules, conjugaison, etc.)



Le cycle de traitement d'un corpus de texte

J'en conviens, la partie nettoyage n'est pas la plus intéressante, mais elle est nécessaire. Pour rendre la chose plus attrayante, je vais réaliser une étude partiale et partielle des artistes français de rap et leur vocabulaire propre.

Tout au long de ce chapitre, nous allons utiliser la librairie de traitement de langage [**NLTK**](http://www.nltk.org/) ainsi que les librairies classiques pandas, numpy et scikit. Vous êtes bien entendus libres d'utiliser d'autres librairies et langages qui vous conviennent.

**Récupération du corpus de texte**

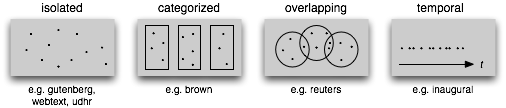
La première étape est la récupération du texte. Il existe plusieurs manières de récupérer du texte : soit depuis une base de données que vous possédez, soit depuis des fichiers XML ou autres que vous possédez, soit en scrapant des pages comme le font les moteurs de recherches, en utilisant une API.

Nous ne traiterons pas cette partie, qui est relativement laborieuse et techniquement peu intéressante. Il existe diverses manières de scraper du texte comme à l'aide des librairies [scrapy](https://scrapy.org/) ou [beautifulsoup](https://www.crummy.com/software/BeautifulSoup/).

Dans mon cas, j'ai scrapé la page wikipédia qui propose une liste des rappeurs français. J'ai ensuite récupéré les paroles des différentes chansons de ces rappeurs sur le site Genius, toujours en scrapant. Je ne suis malheureusement pas autorisé à vous fournir ce jeu de données, libre à vous d'effectuer la même démarche. Vous pouvez aussi utiliser un jeu de données présent par défaut dans la librairie NLTK.

Attention au format d'encodage de vos fichiers texte qui peuvent mener à des erreurs faciles à éviter. On favorisera sur ce cours l'UTF-8 (encodage universel) très utilisé et qui permet l'usage des accents.

Le texte que vous utilisez, que l'on appelle « corpus », peut être organisé de plusieurs manière différentes :



Les différents types de structuration du texte

Chargeons donc les données, dans un dictionnaire python, ce que je fais avec la fonction load\_all\_sentences que j'ai créée pour mon exemple.

db = load\_all\_sentences();

print('chargement de {} vers dans la db'.format(len(db.keys())))

chargement de 788957 vers dans la db

Mon dictionnaire est simplement constitué d'objets de la forme {vers, artiste}

Je crée aussi une table de recherche par artiste car on va s'intéresser à ce qui les différencie et les caractérise. De plus, je veux avoir assez de texte pour chaque artiste donc je vais éliminer ceux qui ont écrit moins de 200 vers.

from collections import defaultdict

base\_artistes = defaultdict(set)

for k,v in db.iteritems():

base\_artistes[v['artistes']].add(k)

artistes = { k:v for k,v in artistes.iteritems() if len(v) > 200 }

print('{} artistes'.format(len(artistes)))

377 artistes

**Exploration du texte : tokenisation et analyse des fréquences**

On veut dans un premier temps étudier le vocabulaire utilisé par chaque artiste. Pour une première intuition, il est judicieux d'observer le nombre de mots utilisés.

On va utiliser la fonction [word\_tokenize](http://www.nltk.org/api/nltk.tokenize.html) (« tokenize » signifie « séparer par mot ») qui va décomposer les vers en tableaux de mots afin de pouvoir effectuer des opérations dessus. Observons déjà son comportement sur un bout de texte simple :

import nltk

test = "Bonjour, je suis un texte d'exemple pour le cours d'Openclassrooms. Soyez attentifs à ce cours !"

nltk.word\_tokenize(test)

['Bonjour',

',',

'je',

'suis',

'un',

'texte',

"d'exemple",

'pour',

'le',

'cours',

"d'Openclassrooms",

'.',

'Soyez',

'attentifs',

'\xc3\xa0',

'ce',

'cours',

'!']

On a bien une séparation par mot. Petit problème en revanche, la ponctuation est conservée comme étant un "token" ! Il faut donc trouver un moyen d'éliminer cette ponctuation, car ce sont les mots qui nous intéressent comme caractéristiques. On remarque aussi qu'il y a un problème sur les apostrophes considérés comme faisant partie du mot. Ainsi "d'exemple" devrait être séparé en "de" et "exemple". Un autre problème, c'est que certains mots ont des majuscules car ils apparaissent en début de phrases ou de vers, alors que ce sont les mêmes mots.

Ça parait tout d'un coup un peu compliqué à mettre en place, n'est-ce pas ? 😏

Les accents sont affichés encodés mais ce n'est pas très grave, on peut revenir à un affichage normal si on le souhaite.

Il est vraiment important de regarder les options sur ce genre de fonctions qui englobent plusieurs actions sur votre corpus afin d'être bien sûr qu'elles effectuent ce que vous voulez.

Le fait d'essayer d'harmoniser les tokens est un processus nommé « **normalisation »**. Bon, on va déjà utiliser les bonnes vieilles expressions régulières pour ne récupérer que les caractères alphanumériques de chaque phrase. Vous trouverez à [cette adresse](https://www.debuggex.com/cheatsheet/regex/python) un rappel utile sur les expressions régulières.

Ensuite, on va utiliser un tokenizer specifique au français ce qui permet de traiter la ponctuation de la bonne manière. On élimine aussi les majuscules peu informatives, avec la fonction « lower ».

tokenizer = nltk.RegexpTokenizer(r'\w+')

tokenizer.tokenize("Bonjour, je suis un texte d'exemple pour le cours d'Openclassrooms. Soyez attentifs à ce cours !")

['Bonjour',

'je',

'suis',

'un',

'texte',

'd',

'exemple',

'pour',

'le',

'cours',

'd',

'Openclassrooms',

'Soyez',

'attentifs',

'\xc3',

'ce',

'cours']

Ah, ça commence à être mieux ! Maintenant qu'on a bien séparé notre texte en unité de mots (tokens) on peut l'appliquer au jeu de données qui nous intéresse, et compter la fréquence d'apparition des différents mots pour avoir une idée du champ lexical. On effectue ce comptage par artiste pour comparer.

tokenizer = nltk.RegexpTokenizer(r'\w+')

def freq\_stats\_corpora():

corpora = defaultdict(list)

# Création d'un corpus de tokens par artiste

for artiste,sentence\_id in artistes.iteritems():

for sentence\_id in sentence\_id:

corpora[artiste] += tokenizer.tokenize(

db[sentence\_id]['text'].decode('utf-8').lower()

)

stats, freq = dict(), dict()

for k, v in corpora.iteritems():

freq[k] = fq = nltk.FreqDist(v)

stats[k] = {'total': len(v)}

return (freq, stats, corpora)

# Récupération des comptages

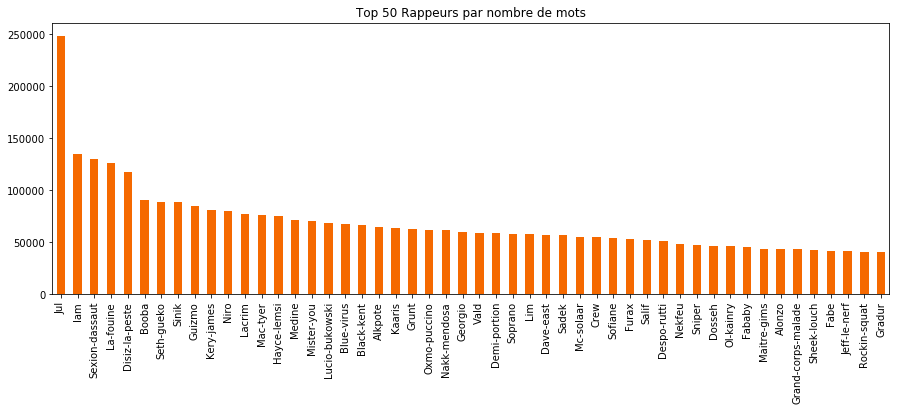
freq, stats, corpora = freq\_stats\_corpora()

df = pd.DataFrame.from\_dict(stats, orient='index')

# Affichage des fréquences

df.sort(columns='total', ascending=False)

df.plot(kind='bar', color="#f56900", title='Top 50 Rappeurs par nombre de mots')



Première tokenisation du corpus

**Nour voyons ici quel artiste a écrit le plus de texte et de chansons**. C'est intéressant, mais qu'en est-il de la variété du champ lexical utilisé, c'est à dire le nombre de mots uniques utilisés par chaque artiste dans leurs chansons ? Nous souhaitons en effet savoir qui a le vocabulaire le plus riche !😀

Pour le savoir, nous devons représenter un document (ou ici, un artiste) par ce qu'on appelle un **bag-of-words.**

Modifions donc notre fonction freq\_stats\_corpora pour faire le comptage du vocabulaire unique.

def freq\_stats\_corpora():

corpora = defaultdict(list)

for artiste,sentence\_ids in artistes.iteritems():

for sentence\_id in sentence\_ids:

corpora[artiste] += tokenizer.tokenize(

db[sentence\_id]['text'].decode('utf-8').lower()

)

stats, freq = dict(), dict()

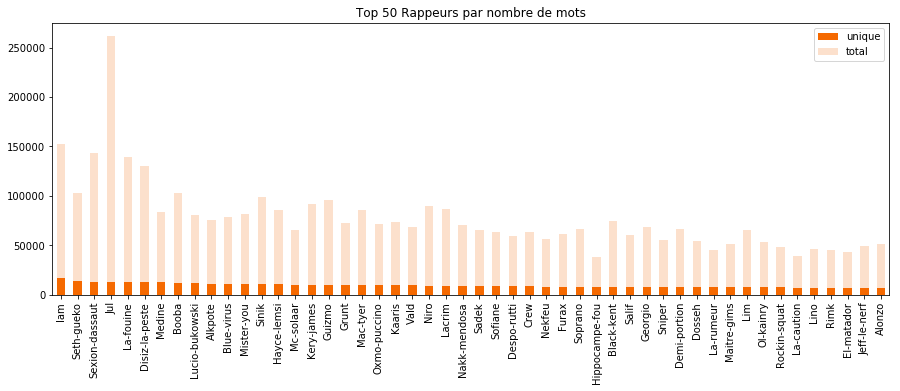
for k, v in corpora.iteritems():

freq[k] = fq = nltk.FreqDist(v)

stats[k] = {'total': len(v), 'unique': len(fq.keys())}

return (freq, stats, corpora)

Affichons à nouveau nos comptages :



Mais ça ne se terminera donc jamais ?! 😢

Pour faciliter des choses, nous décomposons les différentes étapes. À force d'utiliser des corpus de textes, vous saurez les traiter de manière un peu plus automatique en fonction de votre problématique.

Ceci-dit, le prétraitement du texte est **une première étape importante** et il faut vraiment observer le contenu de votre corpus après transformation pour être sûr que les données correspondent à ce que vous désirez, en vue des traitements ultérieurs.

**Conclusion**

Vous possédez à présent une première idée des étapes qui constituent le prétraitement du texte : récupération du corpus, tokenisation et première visualisation des différentes fréquences.

## Nettoyez et normalisez les données

Après la tokenization, voyons comment nettoyer et normaliser notre corpus afin d’obtenir une matrice de vocabulaire et un dictionnaire représentatifs de nos documents.

### 

### Première passe de nettoyage : supprimer les stopwords

La première manipulation souvent effectuée dans le traitement du texte est la suppression de ce qu'on appelle en anglais les stopwords. Ce sont les mots très courants dans la langue étudiée ("et", "à", "le"... en français) qui **n'apportent pas de valeur informative** pour la compréhension du "sens" d'un document et corpus. Ils sont très fréquents et ralentissent notre travail : nous souhaitons donc les supprimer.

Il existe dans la librairie NLTK une liste par défaut des stopwords dans plusieurs langues, notamment le français. Mais nous allons faire ceci d'une autre manière : on va supprimer les mots les plus fréquents du corpus et considérer qu'il font partie du vocabulaire commun et n'apportent aucune information. Ensuite on supprimera aussi les stopwords fournis par NLTK.

Allez, on s'en débarasse !

# Premièrement, on récupère la fréquence totale de chaque mot sur tout le corpus d'artistes

freq\_totale = nltk.Counter()

for k, v in corpora.iteritems():

freq\_totale += freq[k]

# Deuxièmement on décide manière un peu arbitraire du nombre de mots les plus fréquents à supprimer. On pourrait afficher un graphe d'évolution du nombre de mots pour se rendre compte et avoir une meilleure heuristique.

most\_freq = zip(\*freq2.most\_common(100))[0]

# On créé notre set de stopwords final qui cumule ainsi les 100 mots les plus fréquents du corpus ainsi que l'ensemble de stopwords par défaut présent dans la librairie NLTK

sw = set()

sw.update(stopwords)

sw.update(tuple(nltk.corpus.stopwords.words('french')))

Nous avons maintenant le nombre de mots uniques non stopwords utilisés par les artistes. Pour rappel, on souhaite comprendre la variété lexicale des rappeurs choisis. Il est donc logique de supprimer les mots les plus utilisés, ce qui signifie par extension qu'ils ne sont pas porteurs de sens.

On réeffectue notre tokenisation en ignorant les stopwords et on affiche ainsi notre nouvel histogramme des fréquences duquel on a supprimé les stopwords

def freq\_stats\_corpora2(lookup\_table=[]):

corpora = defaultdict(list)

for artist, block\_ids in lt\_artists.iteritems():

for block\_id in block\_ids:

tokens = tokenizer.tokenize(db\_flat[block\_id]['text'].decode('utf-8'))

corpora[artist] += [w for w in tokens if not w in list(sw)]

stats, freq = dict(), dict()

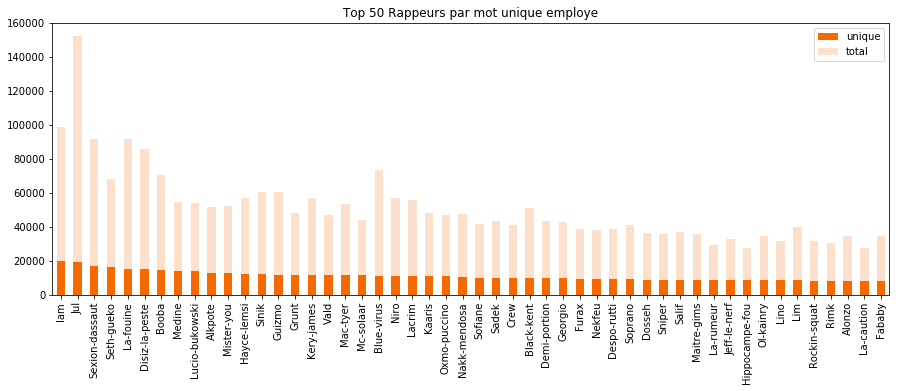
for k, v in corpora.iteritems():

freq[k] = fq = nltk.FreqDist(v)

stats[k] = {'total': len(v), 'unique': len(fq.keys())}

return (freq, stats, corpora)

freq2, stats2, corpora2 = freq\_stats\_corpora2()



Il y a bien un changement dans le classement, maintenant qu'on a enlevé les mots les plus communs, si vous comparez au classement du chapitre précédent.

J'effectue ce classement à titre d'exercice. Il existe de nombreux autres de critères (taille du répertoire, durée de la carrière, etc.) qui ne sont pas pris en compte.

### Deuxième passe : lemmatisation ou racinisation (stemming)

Plus qu'une dernière étape et vous en aurez terminé avec le prétraitement !

Le processus de « lemmatisation » consiste à représenter les mots (ou « lemmes » 😉) sous leur forme canonique. Par exemple pour un verbe, ce sera son infinitif. Pour un nom, son masculin singulier. L'idée étant encore une fois de ne **conserver que le sens des mots** utilisés dans le corpus.

Si l'on reprend notre exemple précédent, "Bonjour, je suis un texte d'exemple pour le cours d'Openclassrooms. Soyez attentifs à ce cours !"

L'idéal serait d'extraire les lemmes suivants : « bonjour, être, texte, exemple, cours, openclassrooms, être, attentif, cours ». Dans le processus de lemmatisation, on transforme donc « suis » en « être » et « attentifs » en « attentif ».

Dans notre cas, je voulais étudier la richesse du vocabulaire des artistes. C'est donc mieux de compter le nombre d'occurrences du verbe être plutôt que de compter séparément chaque usage de conjugaison de ce même verbe. De même pour les pluriels etc. On estime que c'est plus représentatif, j'espère que vous êtes d'accord ! 😏

Il existe un autre processus qui exerce une fonction similaire qui s'appelle la **racinisation**(ou stemming en anglais). Cela consiste à ne conserver que la racine des mots étudiés. L'idée étant de supprimer les suffixes, préfixes et autres des mots afin de ne conserver que leur origine. C'est un procédé plus simple que la lemmatisation et plus rapide à effectuer puisqu'on tronque les mots essentiellement contrairement à la lemmatisation qui nécessite d'utiliser un dictionnaire.

Dans notre cas, on va effectuer une racinisation parce qu'il n'existe pas de fonction de lemmatisation de corpus français dans NLTK 😶 Je suis d'accord que ce serait encore mieux.

from nltk.stem.snowball import FrenchStemmer

stemmer = FrenchStemmer()

def freq\_stats\_corpora3(lookup\_table=[]):

corpora = defaultdict(list)

for artist, block\_ids in lt\_artists.iteritems():

for block\_id in block\_ids:

tokens = tokenizer.tokenize(db\_flat[block\_id]['text'].decode('utf-8').lower())

corpora[artist] += [stemmer.stem(w) for w in tokens if not w in list(sw)]

stats, freq = dict(), dict()

for k, v in corpora.iteritems():

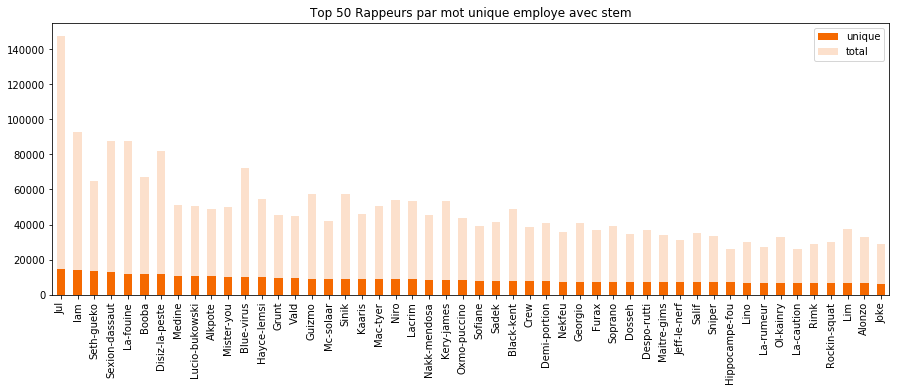
freq[k] = fq = nltk.FreqDist(v)

stats[k] = {'total': len(v), 'unique': len(fq.keys())}

return (freq, stats, corpora)

freq3, stats3, corpora3 = freq\_stats\_corpora3()

df3 = pd.DataFrame.from\_dict(stats3, orient='index').sort(columns='unique', ascending=False)



On a ici utilisé des étapes de nettoyage classique de texte mais en réalité, il est parfois utile de conserver le texte brut quand on est pas dans une recherche du sens d'un document mais de phrases dans leur ensemble puisqu'on cherche le lien entre les différents mots. On aura un aperçu de cet aspect dans les prochains chapitres.

### Conclusion

Vous avez effectué quelques étapes essentielles du prétraitement du texte : tokenisation, suppression des stop-words, lemmatisation et stemming. Nous pouvons maintenant passer à la création de notre ensemble de features représentatives de notre corpus de texte. C'est le sujet de la prochaine partie ! Suivez-moi.

**Entraînez-vous à prétraiter un corpus en vue de créer un moteur de résumés**

**À vous de jouer**

Pour vous entraîner, réalisez cet exercice étape par étape. Une fois terminé, vous pouvez comparer votre travail avec les pistes que je vous propose.

Vous allez nettoyer le texte et de créer un jeu de données d’entraînement en vue de créer un moteur de résumé automatique. Voici le dataset que vous allez utiliser : [https://cs.nyu.edu/~kcho/DMQA/](https://www.google.com/url?q=https://cs.nyu.edu/~kcho/DMQA/&sa=D&ust=1513274358631000&usg=AFQjCNGQ-fD0ocUSLc5xT82rKti4tbU8QA)

**Contexte**

Les données brutes représentent un corpus d’articles CNN récupérés par des chercheurs pour leurs expérimentations. L’objectif est de récupérer les features des documents et les highlights (résumés courts) associés concaténés, en vue d’entraîner un potentiel modèle de création de résumé d’articles.

Article Argentina coach **Alejandro** Sabella believes Lionel Messi’s habit of throwing up during games is because of nerves. The Barcelona star has vomited on the pitch during several games over the last few seasons and appeared to once again during Argentina’s last warm-up match against Slovenia on Saturday….

→

Highlight Argentina coach Sabella believes Messi’s habit of being sick during games is down to nerves.

**Consigne**

Vous devrez effectuer les opérations de traitement suivantes sur le texte, pas forcément dans cet ordre

* Créer des paires de document (article, highlights)
* Suppression de la ponctuation
* Séparation en token en minuscules
* Suppression des stopwords pour les articles
* Calcul des fréquences et tf-idf sur les deux types de documents
* Enregistrement du nouveau jeu de données d’entraînement pour usage ultérieur

Vous êtes allé jusqu'au bout ? Vérifiez-bien que vous ayez les éléments suivants :

* le texte final est tokenisé sans ponctuation et en minuscules a l’aide d’une regex ;
* tous les documents ont été transformés en matrices de valeurs tf-idf calculées sur les paires (articles, highlight).

**Représentez votre corpus en "bag of words"**

Après avoir vu les différents types de nettoyage du texte possible dans les chapitres précédents, nous allons maintenant étudier comment extraire l'information du texte pour le traitement ultérieur par des modèles de machine learning. En d'autres termes, nous cherchons une représentation du langage pour un modèle statistique qui vise à exploiter des données textes.

**Qu'est-ce qu'un « bag of words »**

La manière la plus simple de représenter un document, c'est ce qu'on a effectué dans le chapitre précédent où l'on a considéré tous les mots utilisés pour chaque artiste, sans distinction ni dépendance par vers, chanson, etc. L'analogie est donc qu'on a considéré chaque artiste par la représentation brute d'un "sac" de tous les mots qu'il a utilisé, sans soucis de contexte (ordre, utilisation, etc).

On peut faire la même chose à l'échelle d'un document qu'on représente par un ensemble des mots qu'il contient. En pratique, ça peut être par exemple un vecteur de fréquence d'apparition des différents mots utilisés (ou stem 😉).

Une représentation **bag-of-words** classique sera donc celle dans laquelle on représente chaque document par un vecteur de la taille du vocabulaire |V| et on utilisera la matrice composée de l’ensemble de ces N documents qui forment le corpus comme entrée de nos algorithmes.

**Prendre en compte les co-occurences**

La première chose à considérer, au-delà d'une tokenisation, c'est qu'il est possible de séparer le texte en groupes de plusieurs mots. On appelle les groupes de mots les n-grammes (*n-gram*) : bigrammes pour les couples de mots, trigrammes pour les groupes de 3, etc. Séparer en mot unique est en fait un cas particulier appelé unigrammes.

Par exemple dans la phrase : « Je mange une pomme », on peut extraire les bigrammes {(je, mange), (mange, une) et (une, pomme)}

Pourquoi utiliser des n-grammes avec n>1 ?

Lorsqu'on fait face à une problématique de modélisation du langage, on voit bien que pour étudier idéalement le sens d'un mot il faudrait l'observer dans son contexte. Il existe donc dans un texte (et par extension dans le langage) une forme de dépendance plus ou moins grande entre les mots.

A titre d'exemple, le pronom "je" aura grandement plus de chance d'être suivi d'un verbe. On peut donc traiter chaque mot comme ayant une probabilité d'apparition en fonction du texte qui le précède, c'est à dire comme une séquence. Dans l'idéal, on veut traiter tout le texte de cette façon, mais ce n'est pas possible en termes de capacité de calculs.

En pratique, on peut prendre les quelques mots précédents qui représentent assez d'information pour avoir un modèle séquentiel (markovien) intéressant, d'où l'apparition des n-grammes.

Par exemple on peut assigner une probabilité au bigramme ("je", "mange") :



En pratique, on peut aussi utiliser la fonction bigrams de NLTK

test = "Bonjour, je suis un texte d'exemple pour le cours d'Openclassrooms. Soyez attentifs à ce cours !"

tokens = tokenizer.tokenize(test.lower())

list(nltk.bigrams(tokens))

[(u'bonjour', u'je'), (u'je', u'suis'), (u'suis', u'un'), (u'un', u'texte'), (u'texte', u'd'), (u'd', u'exemple'), (u'exemple', u'pour'), (u'pour', u'le'), (u'le', u'cours'), (u'cours', u'd'), (u'd', u'openclassrooms'), (u'openclassrooms', u'soyez'), (u'soyez', u'attentifs'), (u'attentifs', u'\xe0'), (u'\xe0', u'ce'), (u'ce', u'cours')]

Le modèle de bag-of-words est en fait un cas particulier du modèle n-gram avec n=1

**Une autre manière de pondérer : le tf-idf**

Depuis le départ, on a seulement utilisé les fréquences d'apparition des différents mots/n-grammes présents dans notre corpus. Le problème est que si l'on veut vraiment représenter un document par les n-grammes qu'il contient, il faudrait le faire relativement à leur apparition dans les autres documents.

En effet, si un mot apparait dans d'autres documents, il est donc moins représentatif du document qu'un mot qui n'apparait qu’uniquement dans ce document.

Nous avons d'abord supprimé les mots les plus fréquents de manière générale dans le langage (les fameux *stopwords*). À présent, il ne faut pas considérer le poids d'un mot dans un document comme sa fréquence d'apparition uniquement, mais pondérer cette fréquence par un indicateur **si ce mot est commun ou rare** dans tous les documents.

Pour résumer, le poids du n-gramme est le suivant :



En l’occurence, la métrique tf-idf (Term-Frequency - Inverse Document Frequency) utilise comme indicateur de similarité *l'inverse document frequency* qui est l'inverse de la proportion de document qui contient le terme, à l'échelle logarithmique. Il est appelé logiquement « inverse document frequency » (idf).

Nous calculons donc le poids tf-idf final attribué au n-gramme :



Dans notre exemple, un document égale un artiste. Pour connaître les termes qui représentent le plus un artiste, nous allons utiliser la fonction tf-idf de scikit

import os

def stem\_tokens(tokens, stemmer):

stemmed = []

for item in tokens:

stemmed.append(stemmer.stem(item))

return stemmed

def tokenize(text):

tokens = nltk.word\_tokenize(text)

stems = stem\_tokens(tokens, stemmer)

return stems

for subdir, dirs, files in os.walk(path):

for file in files:

file\_path = subdir + os.path.sep + file

shakes = open(file\_path, 'r')

text = shakes.read()

lowers = text.lower()

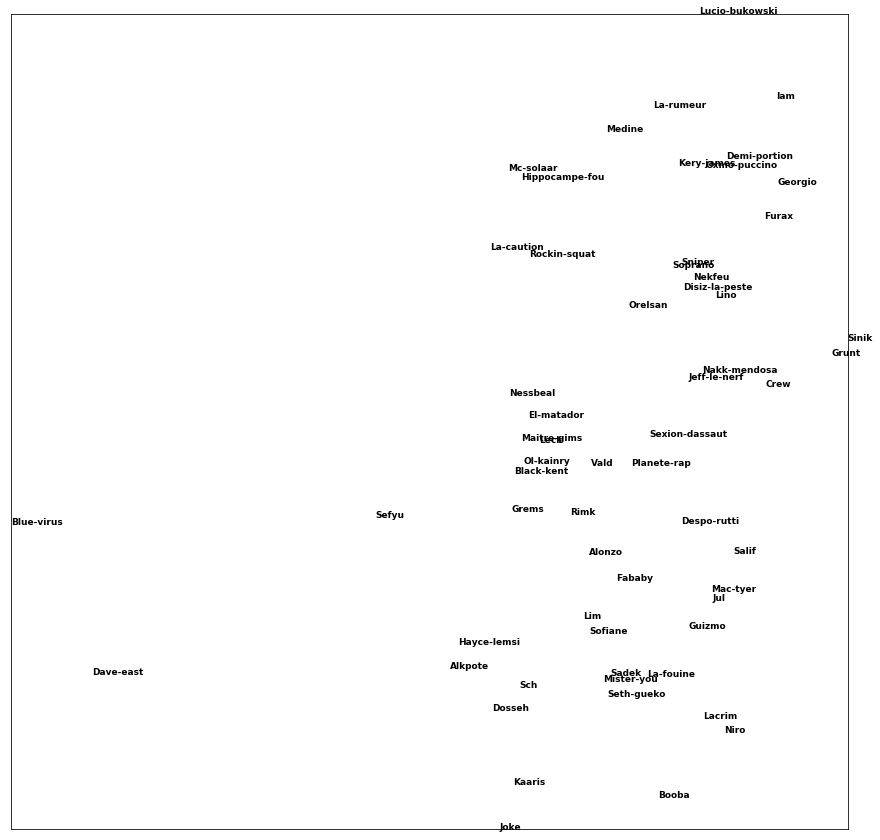
no\_punctuation = lowers.translate(None, string.punctuation)

token\_dict[file] = no\_punctuation

tfidf = TfidfVectorizer(tokenizer=tokenize, stop\_words=sw)

values = tfidf.fit\_transform(token\_dict.values())

Maintenant qu'on a cet indicateur, on peut comparer les différents champs lexicaux qui représentent le plus un artiste. Qui dit comparaison dit similarité. On peut donc utiliser ... t-SNE !



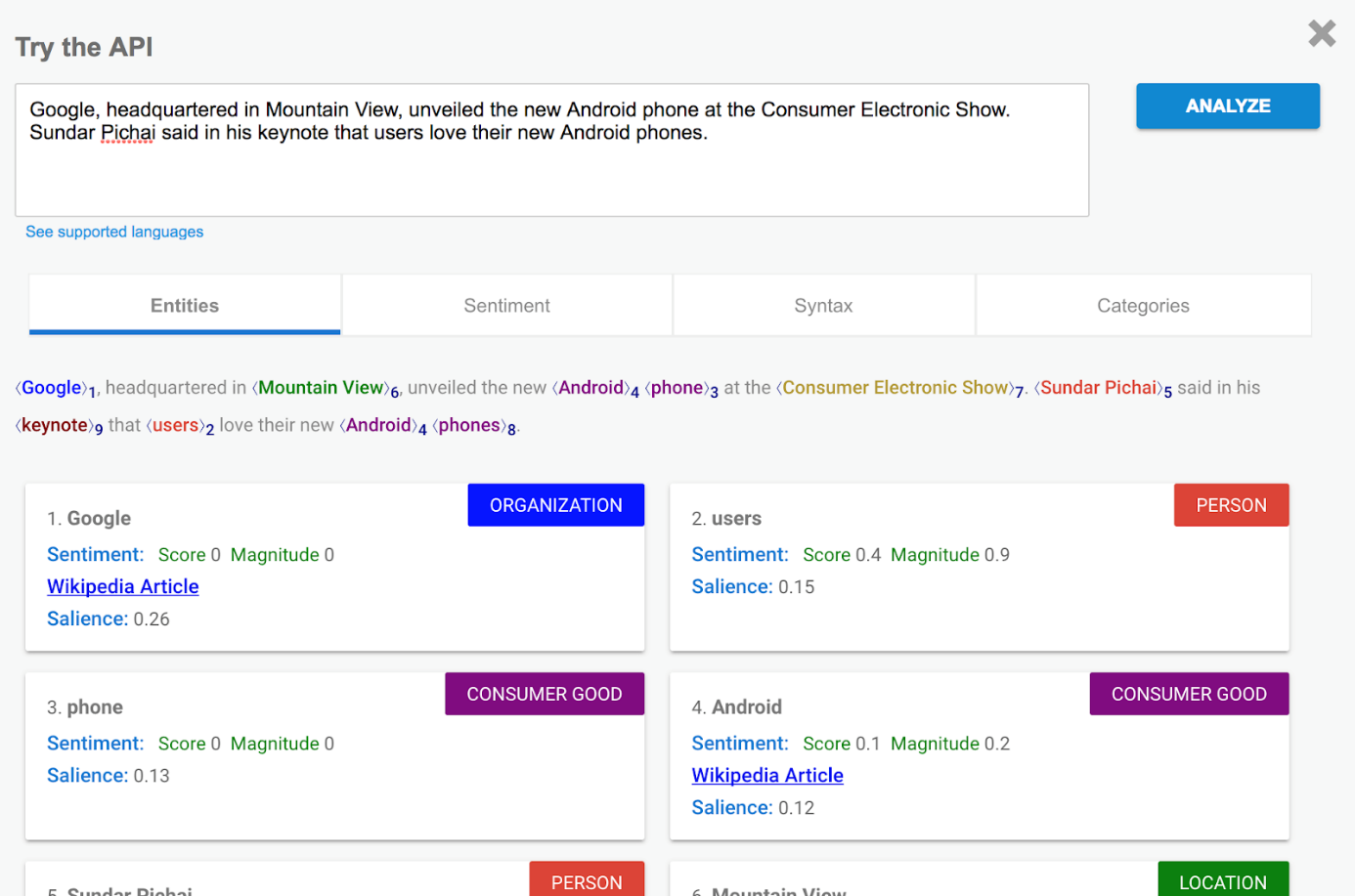
Première visualisation t-sne de notre corpus

**Extraire les informations**

La récupération de caractéristiques va assez loin puisqu'on essaie de dégager de nos documents texte non structurés des informations structurées informatives très restreintes :

* **NER (Named Entity Recognition)** : reconnaître des personnes, endroits, entreprises, etc.
* **Extraction de relations** : essayer d'extraire des relations sémantiques entre différents termes du texte. Par exemple, des relations familiales ("Marie est l'enfant de Patrick") spatiales ("Le piano est devant la fenêtre"), etc. Ces informations peuvent ensuite être stockées dans une base de données relationnelles ou un graphe.
* **Extraction d'événements** : extraire des actions qui arrivent à nos entités. Par exemple "le cours de l'action X a augmenté de 5%" ou bien "le président a déclaré X dans son discours"
* **POS Tagging (Part-of-Speech Tagging)** : représente les méthodes qui récupèrent la nature grammaticale des mots d’une phrase - nom, verbe, adjectif, etc. Ce sont des propriétés qui peuvent servir de caractéristiques utile lors de la création de certains modèles

Ce ne sont pas des méthodes que l’on va traiter dans ce cours. Nous vous donnerons cependant les bases pour pouvoir vous documenter et créer votre propre tagger. Notez qu'en français, il existe peu de ressources toute faites, contrairement à l'anglais où il existe des NER généralistes.



*Vous pouvez essayer l'API Google Natural Langage pour avoir une idée des capacité d'extraction d'information possible par les algorithmes industriels.*

**Attention aux matrices creuses**

Avec les méthodes de comptage évoquées, nous créons en réalité des « matrices creuses ». En effet, les mots ne sont pas présents dans chaque document (le ratio vocabulaire / taille de document est trop élevé). De plus, on utilise plus souvent certains mots (“et”, “le”, etc.) et d’autres plus rarement (dans des contextes précis).

Cette grosse différence créée des matrices larges (de la taille « nombre de documents \* taille du vocabulaire ») qui sont essentiellement vides. On verra qu’on peut utiliser ces matrices avec un certain nombre d’algorithmes, mais c’est tout de même un gaspillage non négligeable de ressources que de travailler avec des matrices de cette taille alors que la plupart des entrées ne sont pas informatives.

De plus, les matrices creuses peuvent biaiser les algorithmes qui considèrent ainsi que les observations à zéro (qui sont présentes en majorité) représente une information à prendre en considération. Si on pense en termes de moyenne par exemple, elle sera écrasée par la présence de toute ces entrées vides sans pour autant apporter plus de sens à notre calcul. Nous allons voir dans les prochaines parties des alternatives de représentation de texte afin de contrecarrer ce problème quand c'est nécessaire.

**Conclusion**

Ces extractions de caractéristiques seront assez intuitives en fonction du problème rencontré. L'idée principale étant de transformer cette masse de texte non structurée en données digestes pour vos algorithmes et vos capacités de calculs. Cependant, ce type de représentation créé des matrices creuses qu’il est parfois difficile à gérer, par exemple dans le cadre de l’utilisation de réseaux de neurones.

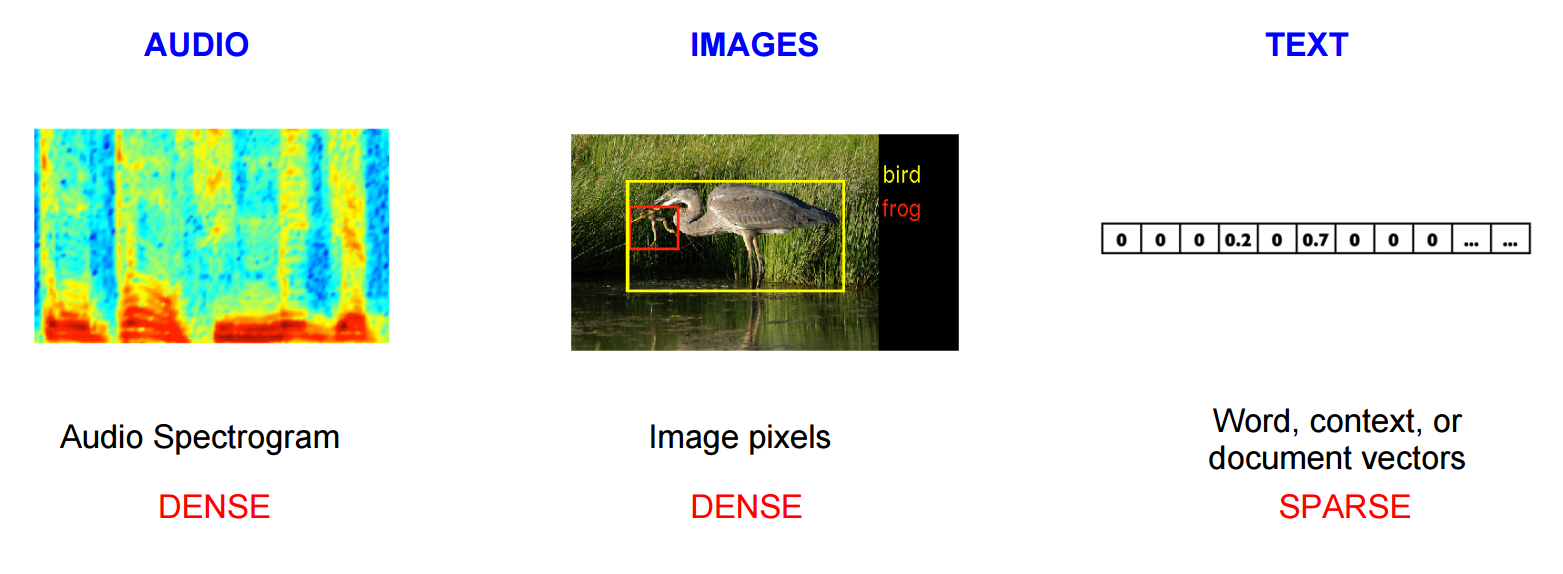
**Effectuez des plongements de mots (word embeddings)**

Les algorithmes et modèles de machine learning utilisent des entrées numériques, puisque nous travaillons avec des espaces topologiques voire métriques la plupart du temps.

Alors comment faire pour représenter du texte… ?

Lorsqu’on traite des données brutes à grande dimensions comme des images ou du spectre audio, on utilise directement des vecteurs de coefficients associées aux intensités de pixel dans le premier cas ou les coefficients de densité spectrales dans le second cas.

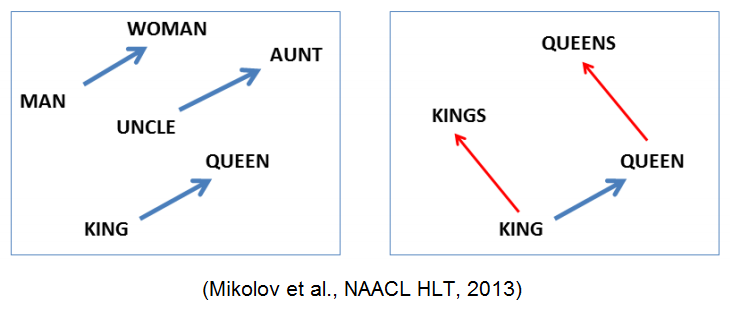
Le problème avec un texte, c’est que l’on va traiter les mots et groupes de mots comme des entités symboliques non porteuses de sens, c’est à dire indépendantes les unes des autres.



On appelle la technique de représentation d’un mot, ou un ensemble de mots en vecteurs de dimension inférieure, *word embeddings,*soit littéralement « plongement de mot ».

Dans les chapitres précédents, nous avons vu une première manière de représenter nos documents comme des bag-of-words ou bag-of-ngrams, essentiellement des méthodes de comptage direct (fréquence ou tf-idf). On a donc techniquement représenté chaque document par un vecteur de fréquences du dictionnaire de mots que l’on avait à disposition. Nous sommes amenés à traiter des données lacunaires ou creuses (en anglais *sparse*). Même avec TF-IDF, on considère des comptages déterministes comme représentatifs de nos données, en l’occurrence la fréquence du mot et la fréquence inverse du document.

Une récente famille de techniques (circa 2013) a permi de repenser ce modèle avec une représentation des mots dans un espace avec une forme de similarité entre eux (c'est-à-dire probabiliste), dans lesquels le sens des mots les rapproche dans cet espace, en termes de distances statistiques. C’est un plongement dans un espace de dimension inférieur autour de 20-100 dimensions généralement. Son petit nom : word2vec.

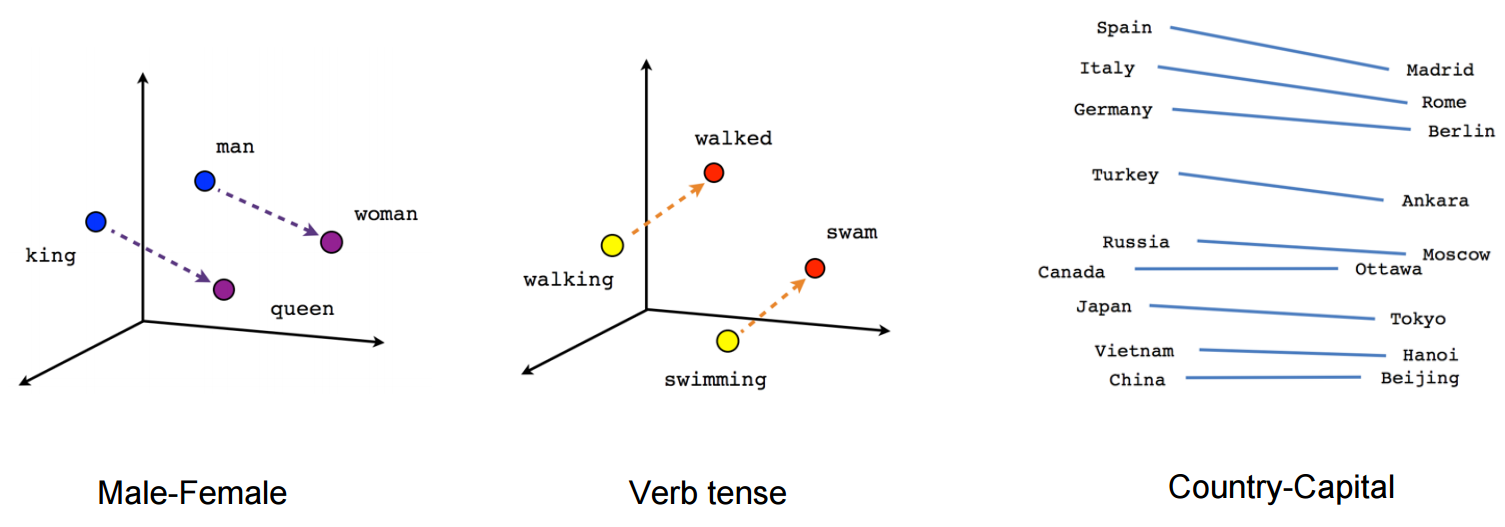


Mais comment trouver le sens des mots ? C’est un peu vague comme concept, non ?

Effectivement. L’hypothèse principale de ces méthodes étant de prendre en compte le “contexte” dans lequel le mot a été trouvé, c’est à dire les mots avec lesquels il est souvent utilisé. On appelle cette hypothèse *distributional hypothesis.*

Et ce qui est intéressant, c’est que ce contexte permet de créer un espace qui rapproche des mots qui ne se sont pas forcément trouvés à côté les uns des autres dans un corpus ! Ces méthodes de représentation vectorielles ont aussi permis d’entraîner des modèles de représentation des mots sur des corpus beaucoup plus grands (des centaines de milliards de mots par exemple...)

Ces représentations possèdent des capacités surprenantes. Par exemple, on peut retrouver beaucoup de régularités linguistiques simplement en effectuant des translations linéaires dans cet espace de représentation. Par exemple le résultat de vec(“Madrid”) - vec(“Spain”) + vec(“France”) donne une position dont le vecteur le plus proche est vec(“Paris”) !

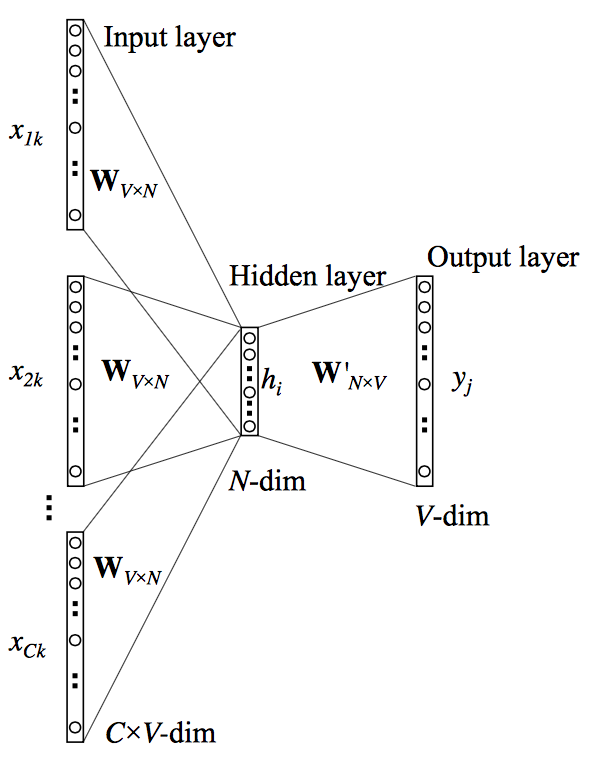


Afin de calculer les vecteurs qui représentent les mots, les méthodes word2vec utilisent des perceptrons linéaires simples avec une seule couche cachée. L’idée est de compresser notre corpus vers un dictionnaire de vecteurs denses de dimension bien inférieure choisie.

Je n’ai pas l’impression que le contexte a été pris en compte, si ?

On ne va pas détailler les méthodes d’entraînement en détails, mais il faut savoir qu’il en existe deux principales. La première appelée « Continuous Bag of Words » (CBOW), qui entraîne le réseau de neurones pour prédire un mot en fonction de son contexte, c’est à dire les mots avant/après dans une phrase. Dans la seconde méthode, on essaie de prédire le contexte en fonction du mot. C’est la technique du « skip-gram ».

En d’autres termes, l’entrée du réseau de neurones dans le cadre du CBOW prend une fenêtre autour du mot et essaie de prédire le mot en sortie. Dans le cadre du skip-gram on essaie de faire l’inverse, c’est-à-dire prédire les mots autour sur une fenêtre déterminée à l’avance à l’aide du mot étudié en entrée.

Représentation du modèle CBOW

Et c’est cette hypothèse - le fait que les mots soient caractérisés par les mots les entourant - qui permet de créer cette compression. Des mots davantage associés aux mêmes mots seront proches dans l’espace d’arrivée.

**Utiliser un modèle de langage**

Avoir cette représentation vectorielle des mots permet d’utiliser ces vecteurs comme des features dans un grand nombre de tâches de base de traitement du langage. On peut ainsi alimenter des algorithmes classiques tels qu’un SVM ou un réseau de neurones avec nos vecteurs caractéristiques des mots.

Pour récapituler, on peut transformer un texte en ses features soit :

* En utilisant une représentation de comptage creuse - fréquence d’apparition du mot dans un document, ou vecteur tf-idf d’un document, etc.
* En utilisant une représentation type word2vec dense - dans laquelle le mot possède une représentation dans un espace qui le positionne en fonction des mots adjacents

Nous verrons dans les prochains chapitres comment utiliser ces représentations de documents dans nos algorithmes afin de pouvoir effectuer différentes tâches de classification, modélisations non supervisées et autres manipulations propres au traitement de langage.

**Alternatives à word2vec**

En pratique, on peut utiliser d’autres types de représentations denses des mots, au-delà du choix de l’algorithme (CBOW, skipgram) et de la dimension. Il existe notamment d’autres méthodes de plongement (word embeddings) tels que [gloVe](https://nlp.stanford.edu/projects/glove/) et [FastText](https://fasttext.cc/).

[Certains favorisent](http://multithreaded.stitchfix.com/blog/2017/10/18/stop-using-word2vec/) même l’utilisation d’une simple décomposition SVD sur une matrice PMI (pointwise mutual information) qui donneraient des performances largement suffisantes pour la plupart des applications industrielles.

L’idée fondamentale reste la même : compresser de manière non supervisée la représentation d’un mot à partir d’un gros corpus de texte représentatif de votre langage, afin d’obtenir un vecteur dense qui permet de visualiser et de fournir à nos algorithmes des features plus intéressantes.

**Entraîner son propre embedding**

Il existe plusieurs plongements disponibles directement en ligne (surtout en anglais) qui ont été entrainés avec les méthodes évoquées plus haut. C’est utile pour avoir un modèle de représentation très général de votre vocabulaire sur une langue en particulier. Cependant vous êtes limité au registre du corpus utilisé. Par exemple si l’embedding a été réalisé sur Wikipédia, vous allez avoir un registre relativement soutenu. Cela biaise un peu la modélisation et surtout rend moins précis votre plongement vis à vis du problème rencontré. Cela a aussi pour effet de dissiper des différences entre vecteurs qui pourraient être utiles pour votre problème en faveur des grandes disparités (comme l’exemple masculin / féminin ou capitale / pays).

L’avantage d’entraîner son propre embedding est donc d’avoir un plongement spécifique à notre corpus et donc plus performant concernant la problématique que l’on veut traiter. Vous pouvez ensuite comparer cet embedding à une baseline utilisant un embedding général.

Vous pouvez par exemple entraîner votre propre embedding word2vec en suivant ce tutorial : [**https://radimrehurek.com/gensim/auto\_examples/**](https://radimrehurek.com/gensim/auto_examples/)

Le problème qui en découle est bien sûr un manque de généralisation si le jeu de données utilisé n’est pas représentatif de la population totale.

**Conclusion**

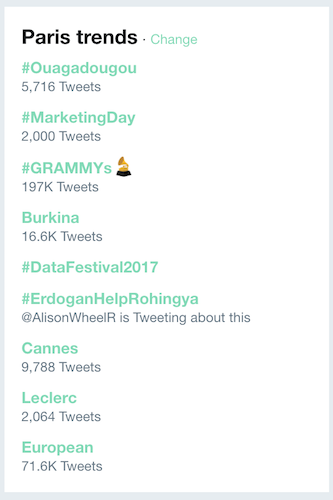
Les plongements de mots sont les représentations les plus utilisées actuellement dans les dernières méthodes de traitement du langage. C'est devenu un outil incontournable à tester lors de vos manipulations de texte. Si vous avez le temps et les ressources nécessaires, n'hésitez pas à entraîner votre propre embedding spécialisé sur votre problématique.

## Modélisez des sujets avec des méthodes non supervisées

Dans le monde actuel, où la quantité de texte non structuré, augmente drastiquement (commentaires, articles de blog, etc.), il serait vraiment utile d’avoir des outils qui permettent de structurer automatiquement l’information, de manière à pouvoir rapidement accéder à ce qui nous intéresse, filtrer le bruit mais aussi détecter l’apparition de nouveau sujet d’intérêts.

On peut retrouver cette nécessité à plusieurs niveaux dans les applications :

* Détecter les fameux “trending topics” de Twitter
* Trouver les nouveaux sujets d’informations abordés par les médias
* Détecter un nouveau stock intéressant d’après un groupement de textes d’experts
* Organiser un corpus de textes scientifiques autour des thématiques abordées
* Trouver les différents aspects d’un produit abordés par des commentaires afin de pouvoir plus facilement l’améliorer à partir de feedback utilisateur



Les « trending topics » de Twitter

C’est dans ce cadre qu’intervient la modélisation de sujets (topic modeling en anglais) qui représente le spectre des différentes approches permettant cette détection.

Dans ce chapitre, on va étudier les plus populaires, afin d’avoir une intuition de cette famille d’algorithmes. Il faut savoir, en revanche, qu’il reste difficile d’appliquer directement ces algorithmes à toutes les situations et qu’il existe un grand nombre de variantes spécifiques à des problématiques plus précises qui correspondront à ce que vous recherchez.

Il faut donc se documenter et comprendre les différents critères différenciants (e.g. modélisation dynamique des sujets dans le temps, longueur du document, nombre de sujets abordés, etc.) qui vous permettront d’effectuer un choix informé.

C’est aussi une famille de méthode utilisé essentiellement en exploration voire semi-supervisée, c’est à dire qui permet de détecter si effectivement il y a de grandes catégories abordées, et ensuite les affiner lors du passage en production, et supervision des nouveaux documents entrants.

Pour résumer, la modélisation automatique de sujet permet de détecter les sujets latents abordés dans un corpus de documents, assigner les sujets détectés à ces différents documents. On peut ensuite utiliser ces sujets pour effectuer des recherches plus rapides, organiser les documents ou les résumer automatiquement.

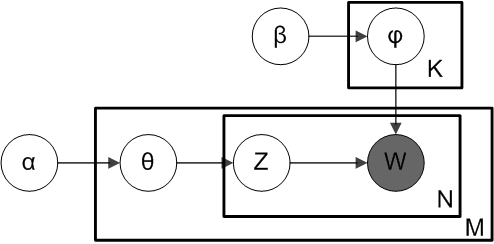
### Première intuition : Latent Dirichlet Allocation (LDA)

La première méthode vraiment efficace est nommée **LDA** (Latent Dirichlet Allocation). C’est une méthode non-supervisée générative qui se base sur les hypothèses suivantes :

* Chaque document du corpus est un ensemble de mots sans ordre (bag-of-words) ;
* Chaque document mm aborde un certain nombre de thèmes dans différentes proportions qui lui sont propres p(θm) ;
* Chaque mot possède une distribution associée à chaque thème p(ϕk). On peut ainsi représenter chaque thème par une probabilité sur chaque mot.
* zn représente le thème du mot wn

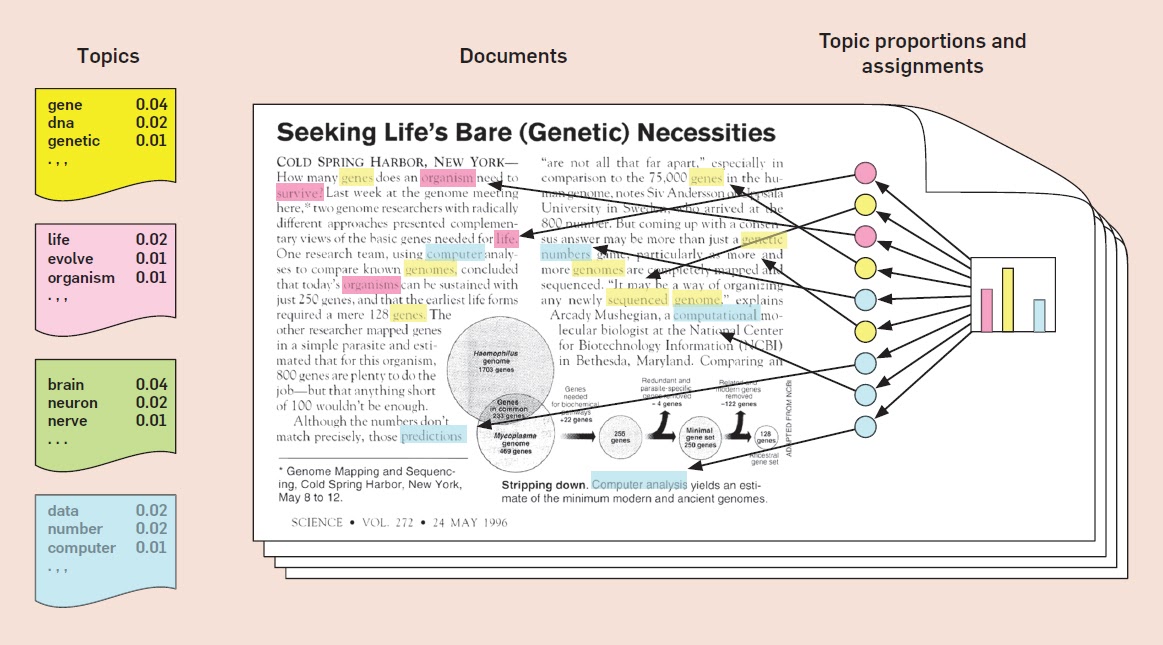
Puisque l'on a accès uniquement aux documents, on doit déterminer quels sont les thèmes, les distributions de chaque mot sur les thèmes, la fréquence d’apparition de chaque thème sur le corpus.

Une représentation formelle sous forme de modèle probabiliste graphique est la suivante :



Pour rappel, un modèle génératif définit une probabilité de distribution jointe sur les différentes variables identifiées, à la fois observées et latentes. Une fois ces variables identifiées, ainsi que les différentes probabilités de distribution associées, l’objectif est de retrouver par exemple les distributions latentes par rapport aux variables observées.

Dans notre cas, nous souhaitons retrouver les distributions de mots sur les différents thèmes, les différentes proportions de thèmes pour chaque document, les proportions d’apparition d’un thème sur le corpus. Tout cela à partir des différents documents. Ce qui nous permet par la suite de déterminer le thème d’un document, les mots les plus associés à certains thèmes, etc.



La **loi Dirichlet** est la conjuguée de la loi multinomiale, ce qui signifie qu’elle s’accorde bien avec cette loi en tant que distribution a posteriori en termes de factorisation. On utilise ainsi la distribution de Dirichlet sur la proportion globale des thèmes ainsi que sur chaque distribution de thèmes sur les mots.

Comme nous l'avons dit en introduction, ce modèle représente une version de base assez générale de la modélisation d’un corpus de documents. Il est possible d’ajouter des hypothèses supplémentaires sur la structure des données afin de capturer une plus grande partie de l’organisation de l’information. A titre d’exemple, voici une liste des hypothèses supplémentaires qui mènent à une modélisation plus riche :

* **La distribution des mots sur les thèmes évolue avec le temps**. Ce qui signifie qu’il faut créer une séquence de distribution pour chaque thème qui permet de modéliser l’évolution du thème dans le temps.
* **Certains thèmes sont plus proches que d’autres**. L’hypothèse d’utiliser la distribution de Dirichlet considère que les différents thèmes sont complètement indépendants alors qu’en réalité certains thèmes ont en général plus de chances d’apparaître ensemble.

### Laissez la partie inférence aux librairies !

Effectuer l’inférence de ce modèle est relativement complexe techniquement. Il faut notamment passer par des approximations et des algorithmes qui simplifient le modèle afin de pouvoir le calculer (par exemple mean field ou gibbs sampling). Dans tous les cas on va utiliser des packages tout fait afin de travailler sur nos données.

Les plus connus sont déjà intégrés directement dans les librairies (scikit implémente une version de LDA) mais il faudra par la suite effectuer vos propres recherches afin de trouver des implémentations que vous pourrez utiliser dans des cas plus précis.

### Utiliser LDA sur un cas pratique : Le newsgroup dataset

Afin d’illustrer le type de retours que l’on peut avoir avec ce genre de méthodes, on va appliquer l’algorithme du LDA sur un dataset classique déjà présent dans la librairie scikit : le newsgroup dataset qui regroupe un ensemble de 20,000 documents articles d'actualité.

#### Prétraitement

from sklearn.datasets import fetch\_20newsgroups

dataset = fetch\_20newsgroups(shuffle=True, random\_state=1, remove=('headers', 'footers', 'quotes'))

documents = dataset.data

#### Créer le modèle LDA

from sklearn.decomposition import LatentDirichletAllocation

n\_topics = 20

# Créer le modèle LDA

lda = LatentDirichletAllocation(

n\_components=n\_topics,

max\_iter=5,

learning\_method='online',

learning\_offset=50.,

random\_state=0)

# Fitter sur les données

lda.fit(tf)

#### Evaluation

On affiche les mots les plus représentatifs des sujets modélisés, afin de nous donner une idée de leur signification et voir si effectivement on trouve des catégories claires pour les humains et représentatifs de notre corpus.

def display\_topics(model, feature\_names, no\_top\_words):

for topic\_idx, topic in enumerate(model.components\_):

print("Topic {}:".format(topic\_idx))

print(" ".join([feature\_names[i] for i in topic.argsort()[:-no\_top\_words - 1:-1]]))

no\_top\_words = 10

display\_topics(lda, tf\_feature\_names, no\_top\_words)

Topic 0: government people mr law gun state president states public use

Topic 1: drive card disk bit scsi use mac memory thanks pc

Topic 2: said people armenian armenians turkish did saw went came women

Topic 3: year good just time game car team years like think

Topic 4: 10 00 15 25 12 11 20 14 17 16

Topic 5: windows window program version file dos use files available display

Topic 6: edu file space com information mail data send available program

Topic 7: ax max b8f g9v a86 pl 145 1d9 0t 34u

Topic 8: god people jesus believe does say think israel christian true

Topic 9: don know like just think ve want does use good

Comme on peut le voir, quelques sujets qui ont été modélisés sont effectivement interprétables : le sujet 4 représente simplement les nombres. Le sujet 5 représente globalement l’informatique. Le sujet 8 semble représenter la religion, etc etc.

#### Une alternative, NMF

Un autre type de modélisation de sujet automatique non supervisée est NMF (Negative Matrix Factorisation).

from sklearn.feature\_extraction.text import TfidfVectorizer, CountVectorizer

from sklearn.decomposition import NMF

# NMF is able to use tf-idf

tfidf\_vectorizer = TfidfVectorizer(max\_df=0.95,

min\_df=2,

max\_features=no\_features,

stop\_words='english')

tfidf = tfidf\_vectorizer.fit\_transform(documents)

tfidf\_feature\_names = tfidf\_vectorizer.get\_feature\_names()

no\_topics = 20

# Run NMF

nmf = NMF(n\_components=no\_topics, random\_state=1, alpha=.1, l1\_ratio=.5, init='nndsvd')

nmf.fit(tfidf)

no\_top\_words = 10

display\_topics(nmf, tfidf\_feature\_names, no\_top\_words)

Topic 0: people don think like know time right good did say

Topic 1: windows file use dos files window using program problem card

Topic 2: god jesus bible christ faith believe christian christians church sin

Topic 3: drive scsi drives hard disk ide controller floppy cd mac

Topic 4: game team year games season players play hockey win player

Topic 5: key chip encryption clipper keys government escrow public use algorithm

Topic 6: thanks does know mail advance hi anybody info looking help

Topic 7: car new 00 sale price 10 offer condition shipping 20

Topic 8: just like don thought ll got oh tell mean fine

Topic 9: edu soon cs university com email internet article ftp send

### 

### Conclusion

L'objectif de ce type de modélisation de sujets est de récupérer de potentielles catégories pour des traitements ultérieurs. Cette modélisation offre surtout une meilleure compréhension de la structuration du texte en vue de création de features manuelles (mettre l'accent sur certains mots, comprendre ce qui définit une catégorie, etc.)

# Partie 2

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Classifier des textes avec une LDA
* Effectuer des plongements de mots
* Représenter un corpus textuel en bag-of-word

### Question 1

**Un plongement de mots (ou word embedding) permet de transformer un mot en un vecteur**

* + 

dense

* + 

creux

* + 

ça dépend des cas

*L'objectif des plongements de mots est de transformer les mots en vecteurs denses afin de traiter des problèmes avec des dimensions plus petites et ne pas écraser les calculs à cause du travail sur des matrices creuses.*

### Question 2

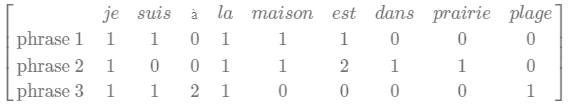
**Quelle est la représentation bag-of-words du corpus de 3 documents suivants ? On représente chaque mot du vocabulaire dans leur ordre d'apparition et chaque ligne représente un document.**

{"Je suis à la maison", "La maison est dans la prairie", "Je suis à la plage"}

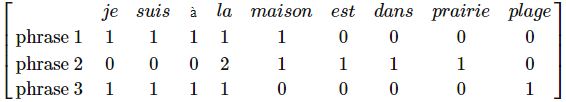
* + 



* + 



* + 



*C'est un simple comptage par document du nombre d'occurrence des mots. Chaque ligne représente le comptage par document. Chaque colonne représente un mot dans l'ordre de leur apparition*

### Question 3

**La matrice TF-IDF est définie pour chaque mot relativement à un corpus, comme le produit TF \* IDF où:**

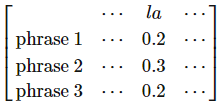
* + TF = nombre de fois où le mot est dans le document / nombres de mots dans le document
  + IDF = nombre de document / nombre de documents où apparaît le mot

**Considérons le mot "la" dans le corpus**

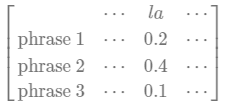
{"Je suis à la maison", "La maison est dans la prairie", "Je suis à la plage"}

**Quelle est la bonne colonne dans la matrice TF-IDF pour ce mot**

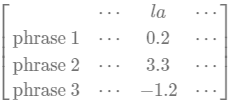
* + 



* + 



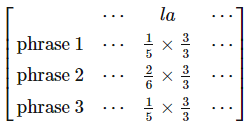
* + 



*La première réponse est correcte.*

* + *On a 3 documents dans le corpus et le mot "la" est présent dans les 3 donc IDF = 3/3*
  + *Les phrases 1 et 3 ont 5 mots et la phrase 2 a 6 mots*
  + *"la" est present 1 fois dans les phrases 1 et 3 et 2 fois dans la phrase 2*

*D'où*



*À noter que la matrice TF-IDF ne contient jamais de valeur négative ou supérieure à 1.*

### Question 4

**L'algorithme LDA permet de**

*Attention, plusieurs réponses sont possibles.*

* + 

Prendre en compte la distribution de chaque sujet sur les documents

* + 

Prendre en compte la distribution de chaque mot par sujet

* + 

Prendre en compte la distribution de chaque mot par document

*La distribution de chaque mot par document n'est pas déterminée directement par le document en cours mais par les proportions de sujets associées à ce document*

### Question 5

**On peut utiliser l'algorithme LDA pour**

*Attention, plusieurs réponses sont possibles.*

* + 

Trouver quels sont les différents sujets abordés dans les documents

* + 

Trouver le nombre de sujets abordés dans le corpus

* + 

Trouver la proportion d'un mot sur un des sujets détectés

*Le nombre de sujets reste un hyperparamètre pour LDA à trouver au préalable*

### Question 6

**CBOW et skip-gram sont deux méthodes qui permettent de créer un modèle de plongement de mots. La différence entre les deux réside dans le fait que**

* + 

on prend en compte un nombre de mots à proximité différents

* + 

on ne considère pas le même type de proximité entre les mots

* + 

on prend en entrée les mots alentours dans un cas, en sortie dans le second cas

* + 

on traite des n-grammes de taille différentes dans les deux cas

*L'entraînement de ces méthodes est basiquement de trouver des représentations vectorielles qui modélisent la proximité entre les mots, par leur utilisation dans des phrases. On considère dans un premier cas de modéliser les mots à proximité à partir du mot considéré en entrée, tandis qu'on essaie de retrouver les mots à proximité à partir du mot considéré dans le second cas.*

### Question 7

**Combien y-a-t-il de trigrammes dans la phrase ci-dessous ?**

La seconde partie de du cours de traitement de texte traite de la transformation des données textuelles

* + 

17

* + 

18

* + 

15

* + 

16

*On peut faire un trigrammes qui commencent par tous les mots de la phrase sauf les deux derniers donc (nombre de mots - 2)  = 15*

### Question 8

**Dans le cas où on aurait bien entraîné un plongement de mots, les opérations : boy - son + daughter = ?**

* + 

gender

* + 

girl

* + 

aucun des deux

*Les plongements de mots (ou word embeddings) transforment des relations entre les mots en relations algébriques très simplement interprétable. En l'occurence ici, on se déplace du point girl par la translation de vecteur**. On retrouve donc la même relation équivalente boy <> son et girl <> daughter*

### Question 9

**Globalement, les méthodes non supervisées sur du texte vont permettre de**

*Attention, plusieurs réponses sont possibles.*

* + 

Trouver des représentations utiles du corpus, par exemple des sujets latents faisant office de catégories

* + 

Créer des features utiles pour modélisation ultérieure

* + 

Normaliser le texte automatiquement

*La normalisation du texte n'est pas une procédure qui peut s'effectuer directement avec des méthodes non-supervisées et requiert un certain nombre d'étapes à effectuer à la main selon la problématique rencontrée.*

## Opérez une première classification naïve de sentiments

Dans cette partie, nous allons traiter le problème de la classification supervisée, et voir comment utiliser les features que nous avons créées pour alimenter nos algorithmes d’apprentissage, essentiellement dans le cadre de la catégorisation de texte.

Cette partie ne traitera pas le problème de classification plus général qui s’applique à plusieurs niveaux dans un texte : prédire quel sera le prochain mot si on considère chaque mot comme une catégorie. Ou encore le POS tagging qui consiste à considérer la catégorie grammaticale d’un mot (verbe nom etc).

Mais attends, on a vraiment besoin d’algorithme d’apprentissage supervisé pour catégoriser du texte ? On ne peut pas juste assigner quelques mots-clés à des catégories, et hop ?

Vous faites référence aux « systèmes experts » (des règles de fonctionnement codées à la main) qui ne sont malheureusement pas assez flexibles dans la plupart des cas qui nous intéressent et demandent beaucoup de maintenance, qui va en grandissant avec l’échelle de nos données et le nombre de classes à traiter. Cela reste cependant une alternative valable et robuste dans certains cas, à ne pas négliger automatiquement.

### Petit rappel de la méthodologie

La première question à se poser est : qu’est-ce que l'on veut accomplir ?

Dans cette partie, nous allons nous concentrer sur le problème de classification : **pouvoir assigner une catégorie à un document texte fourni en entrée**. Par exemple, un type d’actualité associé à un article, les catégories d’une page Wikipédia, le tag associé à un tweet, etc.

Une application spécifique au langage est la détection de sentiment/émotion d’un document. Cela peut être binaire (positif ou négatif), ou avec plus de nuances (triste, joyeux, en colère, etc). C’est une forme de classification qui est souvent demandée.

Pour cela on va avoir besoin d’un jeu de données d’entraînement. L’idée, comme d’habitude est qu’il soit équilibré par catégorie et relativement consistant. Cette première étape est essentielle afin de s’assurer du bon fonctionnement du reste des traitements.

La seconde étape sera de faire passer nos données non-structurées de texte à la moulinette que l’on a vue dans la première partie pour en sortir des features utilisables et structurées, par exemple sous forme vectorielle.

Une fois notre jeu de donnée d’entraînement et de test formaté, on va pouvoir appliquer les méthodes classiques de classification (SVM, réseau de neurones, régression logistique) sur ces données afin de résoudre la problématique énoncée.

S’en suit un réglage des hyperparamètres par recherche de grille (ou autre méthodologie) pour améliorer le modèle en fonction de la mesure de performance choisie.

### Rappel sur les algorithmes de classification

Dans ce chapitre, on va se concentrer sur la famille d’algorithme de classification **Naïve Bayes** et sur la **régression logistique**. Ces algorithmes permettent de catégoriser le sentiment d’un texte (positif ou négatif), ou encore si un e-mail est un spam ou non.

Pour rappel, l’algorithme Naive Bayes (multinomial ou non) permet d’effectuer des classifications probabilistes, qui assignent la probabilité d’appartenance à une classe.

C’est un type d’algorithme génératif, où l’on va modéliser chaque classe et essayer de déterminer la probabilité l’appartenance d’une observation à cette classe. A contrario, les algorithmes discriminatifs essaient de comprendre les caractéristiques différenciantes entre les différentes classes possibles.

### Evaluation du sentiment d’un corpus de commentaires Amazon

Comme terrain d’études, nous allons utiliser le jeu de données de commentaire de produits Amazon. L’objectif sera de déterminer si un commentaire est positif ou pas.

#### Petit rappel de classification Naive Bayes

Un peu plus formellement, le problème de classification se traduit ainsi : "trouver la classe 'c' qui a la probabilité la plus grande étant donné le document 'd' fourni" :



On ne sait pas calculer p(c|d) directement, on a donc besoin de simplifier un peu le problème. Ainsi, d’après le théorème de Bayes on a :



L’objectif étant une maximisation sur la classe c, p(d) n’influence pas le résultat. Le problème peut être simplifié :



Comme on l’a vu dans les chapitres précédents, le document 'd' est représenté par un certain nombres de features (les mots qu’il contient), sans conserver l’ordre de ces mots (bag-of-words) et en considérant qu’ils sont indépendant. On a ainsi pour un document de N mots wi



Dans le cadre d’étude de texte, nous travaillons sur des probabilités faibles. Nous allons donc plutôt travailler à l’échelle logarithmique, ce qui ne change rien au problème de maximisation (la fonction log est monotone strictement croissante). Cela nous permet, en bonus, de travailler avec des sommes.



Dans le cadre d’une classification binaire de texte avec unigramme.

La question restante est donc : comment estimer p(c) et p(wi|c) à partir de notre jeu de données d’entraînement. Vous l’aurez deviné, on va utiliser des fréquences :)

La probabilité d’une classe est simplement la fréquence d’apparition de la classe dans le jeu de données d’entraînement :



Et la probabilité d’un mot dans une classe est simplement : la fréquence d’apparition de ce mot dans un type de document par rapport au nombre de mot total dans c.



On lisse cette probabilité pour les mots qui n'apparaitraientt pas dans une classe, ce qui évite de rendre nulle notre fonction de vraisemblance si un mot est à zéro prob (lissage Laplacien) :



#### Testons le modèle

Nous allons utiliser ce jeu de données uniquement pour le test de la détection de sentiment. Avec NLTK, il suffit de charger les données dans un tableau labellisé pour créer notre classifieur, avec chaque mot associé à un booléen confirmant son existence dans le document (en l’occurence ici dans le commentaire).

C'est parti ! Chargeons les données et formatons-les en bag-of-words associé à des booléens :

import nltk

import os

from tools import ap

def format\_sentence(sent):

return ({ word: True for word in nltk.word\_tokenize(sent.decode('utf-8')) })

def load\_training\_set():

training = []

for fp in os.listdir(ap('aclImdb/train/pos')):

example = '{}/{}'.format(ap('aclImdb/train/pos'), fp)

with open(example) as fp:

for i in fp:

training.append([format\_sentence(i), 'pos'])

for fp in os.listdir(ap('aclImdb/train/neg')):

example = '{}/{}'.format(ap('aclImdb/train/neg'), fp)

with open(example) as fp:

for i in fp:

training.append([format\_sentence(i), 'neg'])

return training

training = load\_training\_set()

On peut maintenant entraîner notre classifieur, qui va utiliser les comptages expliqués plus haut pour créer le modèle probabiliste.

from nltk.classify import NaiveBayesClassifier

classifier = NaiveBayesClassifier.train(train)

Je vous invite très fortement à regarder le code d’implémentation du classifieur, voire d’implémenter le vôtre pour comparer les performances et comprendre comment il fonctionne : <http://www.nltk.org/_modules/nltk/classify/naivebayes.html>

Maintenant qu’on a entraîné notre modèle, on peut par exemple observer les features les plus représentatives des classes :

classifier.show\_most\_informative\_features(n=25)

Most Informative Features

Avoid = True neg : pos = 93.4 : 1.0

2/10 = True neg : pos = 75.7 : 1.0

4/10 = True neg : pos = 64.2 : 1.0

\*1/2 = True neg : pos = 57.0 : 1.0

3/10 = True neg : pos = 43.6 : 1.0

Boll = True neg : pos = 37.7 : 1.0

Uwe = True neg : pos = 36.3 : 1.0

7/10 = True pos : neg = 33.2 : 1.0

WORST = True neg : pos = 27.8 : 1.0

8/10 = True pos : neg = 27.5 : 1.0

unwatchable = True neg : pos = 26.7 : 1.0

stinker = True neg : pos = 26.4 : 1.0

Paulie = True pos : neg = 24.3 : 1.0

awful. = True neg : pos = 23.0 : 1.0

1/10 = True neg : pos = 22.7 : 1.0

excellently = True pos : neg = 22.2 : 1.0

MST3K = True neg : pos = 22.1 : 1.0

Capote = True pos : neg = 21.7 : 1.0

awfulness = True neg : pos = 21.0 : 1.0

Highly = True pos : neg = 20.5 : 1.0

dreck = True neg : pos = 19.9 : 1.0

Ajay = True neg : pos = 19.0 : 1.0

complement = True pos : neg = 18.3 : 1.0

incoherent = True neg : pos = 18.2 : 1.0

Prom = True neg : pos = 17.7 : 1.0

Ça parait relativement réaliste sur les ratio - l'utilisation de avoid est très significative d'une review negative pour un ratio d'utilisation de 1 / 93.4 etc

Et les performances de classification sur les données test :

print(accuracy(classifier, test))

0.88754

On voit qu'on arrive à avoir une précision de prédiction de sentiments relativement intéressante (88.75% de précision) pour un premier essai sur le jeu de données test. Pour aller plus loin dans l’amélioration des performances, il peut être judicieux d’effectuer une validation croisée 😉

### Conclusion

L'application d'algorithmes de classification classiques fonctionne dès lors que l'on sait quelles features utiliser à partir de notre corpus de départ.

## Allez plus loin dans la classification de mots

L’utilisation de manière brute d’algorithmes de classification n’est souvent pas suffisante pour obtenir des performances optimales en vue des problèmes auxquels vous serez confrontés. A travers l’exemple de la classification de sentiments, on va voir qu’il est important d’effectuer des modifications préalables de son corpus et de ses features spécifiques à nos problèmes.

### 

### Particularités de la classification de sentiments

Comme la classification des sentiments d’un texte est un problème relativement classique de traitement du langage, il existe un certain nombre de techniques spécifiques qui permettent d’obtenir des modèles plus performants sur cette tâche. C’est une bonne illustration du type de manipulation qui permet d’orienter son travail de modélisation en fonction de la problématique abordée.

Par exemple, on ne va pas utiliser un bag-of-words avec comptage, mais simplement **la présence ou pas d’un mot** dans un document. En effet, que le mot “désastre” apparaisse une ou deux fois influence peu la probabilité que le document soit négatif. C'est un critère flag binaire.

Voici une autre technique utilisée pour régler **le problème de la négation**. Si je dis « je n’ai pas aimé ce film », la négation “n’” **inverse** le sentiment de ma phrase comparé à “aimer” qui se retrouve davantage dans des documents positifs habituellement. Pour contrer ce problème de manière simple, il suffit d’ajouter un indicateur au mot suivant la négation comme signifiant que son sens a été modifié.

Ainsi, une phrase telle que “je n’aime pas cette personne” sera transformée en bag-of-words : { “je”, “ne”, “NON\_aime”, “pas”, “cette”, “personne” }. en ajoutant des préfixes du type "NON\_", une bonne partie des problèmes sur ce genre d’inversions de sens est ainsi réglée.

Enfin, une technique de base est d’utiliser un lexique de mots représentatifs de vos classes, en l’occurrence ici positif et négatif. Si on utilise simplement le nombre d’apparitions de mots du lexique par classe bien construit pour entraîner un modèle de classification, on a déjà une bonne baseline.

Ainsi, toute ces petites modifications et ajouts permettent de rendre notre modèle plus spécialisé pour notre problématique afin de donner de meilleures performances. Je vous encourage à développer ce genre de tweaks lors de la création d’un modèle pour un problème donné.

### Utiliser d’autres types d'algorithmes de classification supervisée

On a utilisé un modèle simple et robuste, le classifieur bayésien naïf. Pourquoi vouloir utiliser un autre type de classifieur ?

Une hypothèse très forte qui est faite lors de l’utilisation du classifieur Bayes est **l’indépendance des features**. Ce qui signifie que si deux features sont en réalité corrélées, elles auront un effet plus fort que ce qu’elles apportent en réalité comme information pour la classification. D’autres types de modèles ne sont pas aussi sensibles à cette corrélation entre les features, voire permettent de modéliser cette interaction entre les features pour rendre le modèle plus performant. C'est une des différences possibles qui influenceront votre choix de classifieur. En général, l'idée est de pré-tester un certain nombre de classifieur qui intuitivement correspondent à votre problème pour savoir sur lequel vous concentrer.

#### Régression Logistique

La régression logistique, à l’opposé de la classification bayes, est un modèle de classification discriminant. Il est expliqué [plus en détail ici](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire/predisez-lineairement-la-probabilite-de-lappartenance-dun-point-a-une-classe).

Son objectif est de maximiser la probabilité d’avoir une classe y = c étant donné certaines features f(x,c) calculées à partir des observations x. Cette maximisation s’effectue en général avec les méthodes classiques de descente de gradient, avec un terme de régularisation supplémentaire.

On peut directement appliquer la régression logistique sur des matrices creuses de la taille du vocabulaire ou sur les vecteurs plus denses que l’on a créé à l’aide des techniques utilisées dans les chapitres précédents.

#### SVM, forêts aléatoires

Le choix d’un classifieur est en fait un retour à l’éternel dilemme biais-variance, indépendamment du fait qu’on est amené à traiter du texte. Le classifieur Naive Bayes possède une variance faible et va pouvoir mieux généraliser plus rapidement, ce qui peut être utile lorsqu’on a un petit jeu de données ou des documents avec peu de texte. La contrepartie, bien sûr, c’est que ce genre de classifieur va avoir une plus faible précision (un biais plus grand) comparé à des classifieurs type SVM + RBF Kernel ou une régression logistique.

Pour en savoir plus sur les avantages et inconvénients de différents classifieurs, notamment appliquées à la classification de sentiments, vous pouvez consulter [**cet article**](https://www.aclweb.org/anthology/P12-2018) simple et efficace. Il s'agit d'un exemple de comparaison qu’il vous sera utile de consulter lors de votre travail préliminaire d’exploration.

### Conclusion

Dans un premier temps, nous favorisions un travail expert sur les features avant de se concentrer sur l'utilisation d'un classifieur approprié à la problématique en cours et aux hypothèses de départ.

## Traitez le corpus de textes à l'aide de réseaux de neurones

Dans ce chapitre, on va s’intéresser à des architectures de réseaux de neurones assez utilisées dans le cadre de tâches de traitement de langage. Ils vont permettre notamment de traiter nos problèmes sous forme de série temporelle (de séquences), ce qui semble adapté à nos applications puisque le texte est une séquence de mots qui sont eux-mêmes une séquence de lettres. 🚀

On traitait déjà dans certains modèles l’aspect séquentiel (par exemple en utilisant des n-grammes > 1) mais l’hypothèse markovienne est très contraignante pour la représentativité du modèle puisqu’on “oublie” de prendre en compte les éléments précédents les éléments conditionnés.

Attention à ne pas plonger dans ces architectures sans tester au préalable des solutions qui sont en apparence plus simple mais beaucoup plus efficace (en termes de temps de calcul notamment) pour la majorité des problématiques que vous rencontrerez. En effet, les débutants ont tendance à tout de suite vouloir se diriger vers ces méthodes qui sont certes extrêmement performantes mais sont difficiles à structurer, entraîner et relativement lente par rapport à des méthodes plus simple qui offrent des performances similaires. L'idée est au moins d'avoir une baseline sur des classifieurs classiques avant de tester des architectures plus complexes !

Ce chapitre a vocation à être une introduction aux (relativement) nouvelles architectures de réseaux de neurones et les méthodes associées. Cependant, je vous invite à vous documenter de manière spécifique sur **votre** problématique afin de pouvoir créer votre modèle. Je présente ici les bases afin que vous puissiez vous débrouiller dans le champ des possibles et pouvoir aborder les ressources avec le bagage nécessaire.

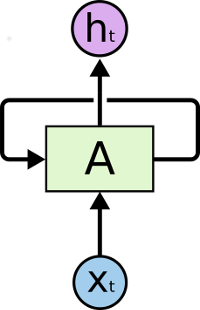
### Représentation des features

On retrouve notamment les plongements (embeddings) en dimension inférieure word2vec (ou autre) comme entrée pour les réseaux de neurones. Ces manipulations préalables ne sont pas inutiles ! Il faut notamment prendre en compte les capacités de calcul et le niveau de représentativité nécessaire dans votre application afin de déterminer la dimension de l’espace d’embedding, par exemple (qui peut varier grosso modo entre 20 et 300 dimensions).

### Différentes architectures possibles

#### L’architecture de base et ses problèmes : le RNN

Les RNN représentent la famille de réseau de neurones qui traite les données de manière séquentielles. L’idée principale étant que chaque nouveau mot qui est prédit à partir de notre modèle, prend en compte l’état précédent afin de s’actualiser. L’état représente fondamentalement l’historique - la mémoire utilisée dans le réseau de neurones pour prendre en compte le passé (TOUT le passé) afin de l’utiliser dans la prédiction à l’instant t.

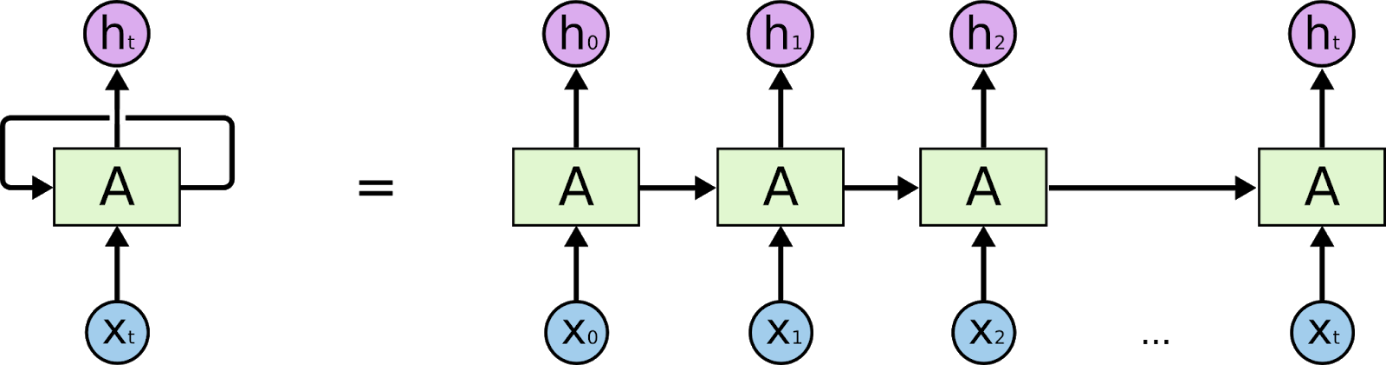


L'architecture du RNN

Dans le cas de tâches associées au texte, on essaie par exemple à chaque étape de prédire le mot en fonction des mots précédents ainsi qu’une observation. Par exemple, le mot dans une langue étrangère dans le cadre d’une traduction.

On a donc une série temporelle qui à chaque instant t, va prendre en entrée une observation ot et l’état précédent xt−1 pour être calculé :





Deux problèmes principaux font suite à cette architecture fondamentale : la **disparition** du gradient, et son **explosion**. En effet lors de la backpropagation, on doit calculer un gradient (en particulier xt/x) qui est sujet à disparition/explosion. Cette disparition notamment a pour effet de faire “oublier” au RNN des informations qui pourraient pourtant être utiles au traitement actuel. Si j’ai un texte dans lequel une information primordiale est présente au début, il est très difficile de créer un RNN qui prenne en compte cette information sur la fin du texte.

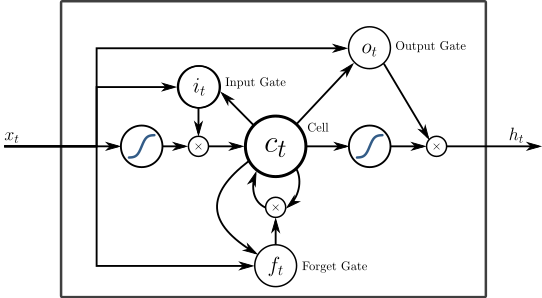
En pratique, on voit que le RNN ne peut prendre en compte que très peu de contexte passé à cause de cette disparition de gradient.

Il existe une grande variété de techniques qui sont utilisés pour mitiger ce problème. Notamment en premier lieu, une architecture plus robuste qui a été imaginée quelques années après l’apparition des RNN appelée **LSTM (Long Short Term Memory)**.

#### LSTM Networks (Long Short Term Memory Networks)

L’idée principale des LSTM est de permettre au réseau “d’oublier” ou de ne pas prendre en compte certaines observations passées afin de pouvoir donner du poids aux informations importantes dans la prédiction actuelle.

Cette idée se traduit par des “gates” qui sont chargées de déterminer l’importance d’une entrée, afin de savoir si on enregistre l’information qui en sort ou pas. Pour une explication détaillée intuitive, rendez-vous sur [cet article de blog](https://colah.github.io/posts/2015-08-Understanding-LSTMs/) qui explique les différentes étapes. En pratique la seule partie qui change vraiment est lors de la mise à jour des réseaux de neurones qui est un peu plus technique.



Une cellule LSTM

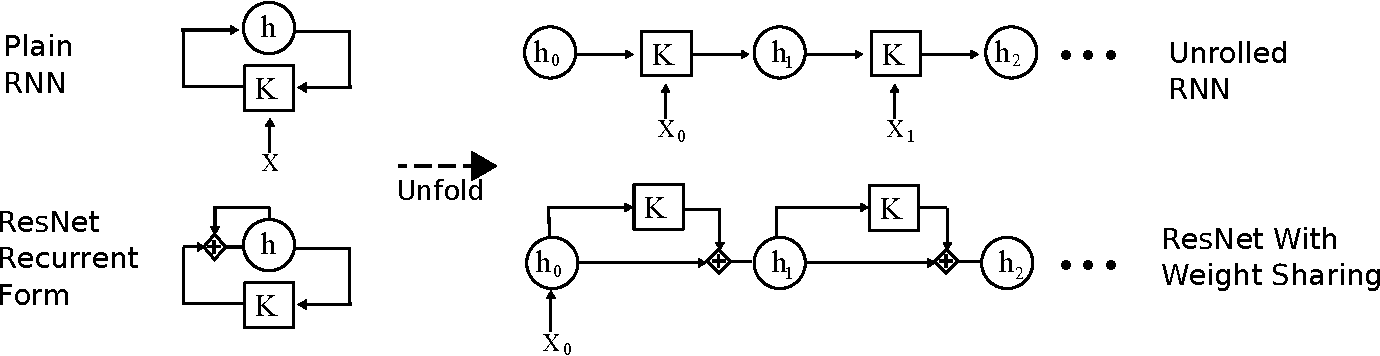
### Etat de l’art des réseaux de neurones en traitement du langage

L’utilisation des réseaux de neurones dans les applications de traitement de langage représente encore un problème ouvert dans la recherche, qui demande à être traité plus en profondeur, notamment sur les problématiques spécifiques à des tâches (quelle architecture correspondent à quelles tâches), des optimisations plus globales et les techniques de transfert d’apprentissage afin de pouvoir utiliser des réseaux déjà entraînés sur de nouveaux problèmes plus facilement.

A titre illustratif, je voudrais citer deux mécanismes qui attirent l’attention depuis quelques temps.

#### Les réseaux de neurones résiduels et autre augmentation de densité

Les réseaux de neurones résiduels sont très utilisés, à la base pour le traitement d’images mais aussi maintenant dans le traitement du texte. Ils permettent aussi de mitiger l’effet de diminution trop brutal du gradient. Pour cela simplement, on ajoute directement l’identité de l’état précédent dans la fonction de calcul. Cela a pour effet de diminuer ce problème de disparition du gradient.



D’autres procédés approfondissent cette idée comme par exemple les [highway networks](https://arxiv.org/pdf/1505.00387.pdf) qui augmentent les fonctions de manière différente mais servent ce même objectif.

#### Les techniques d’attention

L’idée principale des techniques d’attention vise à se concentrer sur une partie de l’information en particulier qui est fournie au réseau de neurones. Ça peut être un mot à traduire au milieu d’une phrase, une partie d’une image lors de la description de celle-ci, etc. Il s'agit donc d’entraîner un réseau de neurones aussi sur un score d’importance à donner à l’ensemble de l’information qui lui est présentée. Ces techniques sont très prometteuses en termes de performance de prédiction.



### Applications

Avec ces architectures, on peut avoir une meilleure représentation d’un corpus de texte puisqu’on prend en compte son caractère séquentiel. Voici quelques domaines d’application dans lesquels ce genre d’architecture a été particulièrement performant :

* Classification de texte
* POS Tagging et NER
* Speech to text
* Traduction automatique
* Génération automatique de texte et notamment de légendes d’images

Ce qu’il faut garder en tête, c’est que les réseaux de neurones récurrents bien entraînés permettent d’intégrer la manière dont est structuré un langage (ou un sous-ensemble du langage) ou toute séquence de texte. Le fait d’avoir ce modèle permet une infinité d’applications liée aux textes, qu’il faut paramétrer et personnaliser pour votre problème spécifique.

Encore une fois, attention à ne pas utiliser un canon pour tuer une mouche, et être bien sûr que toutes les circonstances (taille des données d’entraînement etc) sont présentes pour l’utilisation d’un réseau de neurones efficace.

### Une petite application illustrative : retour aux chanteurs de rap

Afin de démontrer l’efficacité de ces modèles récurrents, essayons de l'appliquer à l'échelle des caractères d’une phrase afin de générer des paroles de rap, dans le style d’un rappeur. Pour cela, on va récupérer [l’implémentation du LSTM de Andrej Karpathy](https://github.com/karpathy/char-rnn)

Si on entraîne sur un corpus de texte particulier (ici le rappeur Nekfeu), voici le texte généré au bout de quelques milliers d'itérations :

Paclois ce b'est un boutes qu'on heêtembarais j'on sa c'est, sout je julam

Dévans

J'etfu l'é-travy pei fais se à jà aura

Qu'le garl

La quincer gé veull'

Atrait

tu vow mon avet peu san enfiiment aure maisas

Et quand j'ta rag, s'lerr

D’est pomps

Fourde bes des quand tout samiectieu ryais

L'heux

Lans l'ai à le gropas, caquis

Je Dacie, à nan poustier juk qu’retape ys qurises suis des conpes émoy ne da tiendir

Plassine grissoble

J'es it-di

J'suis d'acinon puant fet-pes c'tui t'baq

On peut observer notamment la propension à comprendre la longueur d’un vers ou même la taille d'un mot, ce qui montre que le RNN retient que le vers a commencé quelque part avec une majuscule et doit finir par un retour à la ligne. De même, il a compris environ la taille d'un mot et ajoute des espaces dans les bonnes proportions. Il y a donc une forme de mémorisation du contexte passé. On voit aussi des virgules, apostrophes ou des tirets en bonne proportions. L'alternance consoles-voyelles aussi est respectée ainsi que certaines combinaisons caractéristiques ('tt', 'eu', 'es', etc). Quelques mots commencent à être réalistes, mais il n'y a pas assez de données sur le corpus (<250k caractère).

### 

### Conclusion

On a vu ici le type de modélisation possible à l'aide des réseaux de neurones. L'idéal est pour vous d'entraîner votre propre réseau de neurones en vue d'une des tâches applicatives citée plus haut - classification, POS tagging, traduction automatique etc afin de vous rendre compte de la méthodologie d'entraînement sur ce type de problématique. L'entraînement de réseaux de neurones reste pour le moment un processus relativement intuitif avec beaucoup d'heuristiques et de tests nécessaires avant d'obtenir des résultats probants et stables, notamment pour déterminer la valeur des hyperparamètres (nombre & taille des calques, learning rate, algorithme de gradient utilisé, etc)

**Entraînez-vous à classifier du texte**

**À vous de jouer**

Pour vous entraîner, réalisez cet exercice étape par étape. Une fois terminé, vous pouvez comparer votre travail avec les pistes que je vous propose.

Vous utiliserez les données « Reuters Corpus Volume » accessible directement dans scikit learn à l’aide de la fonction *fetch\_rcv1* qui contient 800,000 annonces de presses Reuters étiquetées manuellement. Votre objectif est de réaliser un benchmark de différents types de classifieurs afin de comparer les différentes performances sur ce type de problème.

**Consigne**

Vous devez réaliser les tâches suivantes :

* Charger les données
* Créer différents classifieurs (au moins 3)
* Effectuer une validation croisée sur les différents classifieurs
* Afficher les différentes performances

Le jeu de données est relativement lourd pour un travail en local, avec 650MB compressé de données. Il est conseillé de travailler sur un échantillon dans un premier temps pour s’assurer que tout fonctionne comme prévu pour ensuite traiter tout le jeu de données et obtenir les résultats finaux.

Vérifiez-bien que vous ayez les éléments suivants :

* Au moins 3 classifieurs différents ont été appliqués par validations croisée sur les données correctement, puis les performances ont été évaluées sur chacun.